Combining Discrete and Continuous Domains for SysML-Based Simulation and Test Generation

Jean-Marie Gauthier

SPIM

Thèse de Doctorat

école doctorale **sciences pour l'ingénieur et microtechniques**

UFC UNIVERSITÉ DE FRANCHE-COMTÉ

Combining Discrete and Continuous Domains for SysML-Based Simulation and Test Generation

Unification des ensembles discret et continu pour la simulation et la génération de tests à partir de modèles SysML

JEAN-MARIE GAUTHIER

SPIM

Thèse de Doctorat

UFC

école doctorale **sciences pour l'ingénieur et microtechniques**

UNIVERSITÉ DE FRANCHE-COMTÉ

THÈSE présentée par

JEAN-MARIE GAUTHIER

pour obtenir le

Grade de Docteur de
l'Université de Franche-Comté

Spécialité : **Computer Science**

Combining Discrete and Continuous Domains for SysML-Based Simulation and Test Generation

Unification des ensembles discret et continu pour la simulation et la génération de tests à partir de modèles SysML

Unité de Recherche :
Institut Femto-ST Département d'Informatique des Systèmes Complexes (DISC)

Soutenue publiquement le 19 Novembre 2015 devant le Jury composé de :

PIERRE-ALAIN MULLER	Président du jury	Professeur à l'Université de Haute-Alsace
JEAN-MICHEL BRUEL	Rapporteur	Professeur à l'Université de Toulouse
JORDI CABOT	Rapporteur	Professeur à l'Ecole des Mines de Nantes
FABRICE BOUQUET	Directeur	Professeur à l'Université de Franche-Comté
AHMED HAMMAD	Co-Encadrant	Maître de Conférence à l'Université de Franche-Comté
FABIEN PEUREUX	Co-Encadrant	Maître de Conférence à l'Université de Franche-Comté

ACKNOWLEDGEMENTS

I wish to thank Fabrice Bouquet, Fabien Peureux and Ahmed Hammad for making me the honor of being my supervisors during this thesis and for supporting me and trusted me throughout this adventure. I thank the Regional Council of Franche-Comté, which allowed me to do my PhD through the funding of the SyVAD project.

I thank Jean-Michel Bruel and Jordi Cabot to have agreed to be examiners of my work and for sending me their usefull remarks. Finally, a big thank you to Pierre-Alain Müller for having agreed to chair the jury. Thank you all for the assessment of my work.

I would also like to thank Nathalie Devillers, Marie-Cécile Pera, Dominique Gendreau and Mahmoud Addouche for their cooperation, thus contributing to the achievement of the results presented in this dissertation. I also thank all the members of the DISC department for their welcome and for their sympathy. Thank you to Olga Kouchnarenko for hosting me in the DISC.

I thank my parents who, through their unconditional encouragements, enabled me to achieve this goal. Finally, many thanks to Pauline for her patience and support.

To Pauline and Yohan. . .

CONTENTS

List of Definitions

LIST OF FIGURES

LIST OF TABLES

I

Context, Motivations and State of the Art

1

CONTEXT AND MOTIVATIONS

Contents

This thesis describes an original and innovative modelling approach that uses the System Modeling Language (SysML) [OMG, 2012], to model the System Under Test (SUT) and its environment in the context of *In-the-Loop* design processes. This approach enables validating multi-physical hybrid and safety-critical systems using simulation and automatic functional test generation from models written with SysML. The architecture and the behaviour of the system are described by a SysML model that combines continuous and discrete features for both simulation and testing activities. This approach has been tooled and validated on case studies from research partners of FEMTO-ST[1]. This thesis has been supported by the SyVAD project[2], the Smart Blocks project[3], and the Labex Action[4].

This chapter is organized as follows. Section 1.1 introduces the overall context of the thesis. We present then the involved projects in Sect. 1.2. Section 1.3 describes the issues and the contributions of this work. Finally, after presenting the thesis overview in Sect. 1.4, we detail the running example in Sect. 1.5.

1.1/ CHALLENGES RAISED BY COMPLEX AND CRITICAL SYSTEMS

This section introduces the scientific and industrial challenges about the design, the verification, and the validation of complex and safety-critical systems. Afterwards, background about *In-the-Loop* design processes, modelling, simulation and testing activities are presented. We then present how these activities address the raised challenges and what their limitations are.

[1] http://www.femto-st.fr
[2] http://syvad.univ-fcomte.fr/syvad/ funded by the Regional Council of Franche-Comté.
[3] http://smartblocks.univ-fcomte.fr/ funded by the French National Agency for Research (ANR).
[4] http://www.labex-action.fr/fr

By complex and critical systems, we mean systems that are necessarily composed of physical components, among which some could embed software (System on Chip). Challenges raised by such systems are directly derived from the notions of complexity and criticality.

The complexity of a system may be evaluated by the number of its components and their possible interactions. Concerning physical and embedded systems, the degree of heterogeneity of system components is a major factor of complexity. The heterogeneity of a system could be approximated as the sum of the number of its involved physical and virtual fields. By a physical field, we mean theoretical, physical, and observable (or applicable) knowledge in a particular physical domain such as mechanic, hydraulic, electronic, optic, acoustic, thermodynamic, and so on. By virtual fields, we mean theoretical, numerical, and observable knowledge in a particular computer science domain such as software engineering, optimization, parallel computing, artificial intelligence, and so on.

The separation between physical and virtual components is motivated by the difference between the continuous time paradigm and the discrete time paradigm. On the one hand, physical components are ruled by the laws of physics, i.e. laws that have sense in the theoretical continuous time paradigm. Thus, physical systems, whose state changes continuously over time and whose activity is strongly linked to the time advance, are called continuous (assuming that our real world is continuous). Observations on physical systems can be performed continuously (analogical signal recording). On the other hand, virtual systems are ruled by the theoretical discrete time paradigm. Their state changes at specific time or at specific input values. Indeed, the state change rate of virtual systems may be very high. This allows modern processors to emulate continuous phenomenon (simulation) or to record physical continuous phenomenon (digitizing/sampling signals). While modern processors rely on clocks and bit-sets, observations on virtual systems are possible only at discrete time.

The number of involved components (e.g., 400 km of wiring harness connecting 600 000 signal interfaces in the Airbus A380), the number of physical and virtual domains (e.g., 100 millions of lines of code in the Airbus A380) associated to their time paradigm affect the complexity of a system. As a consequence, interactions between components may become huge or infinite (in the sense of non computable in a reasonable time). That is a major concern especially when the designed system is critical, such as most systems in aeronautical, transportation, nuclear or telecommunication industry. In the field of quality management and functional safety and security, the criticality of a system is defined as the product of the probability of an accident occurrence with its seriousness.

The overall **criticality** of a system could be approximated as the sum of each atomic criticality associated to its safety and security properties. By a safety property, we mean a property that guarantee of being protected from event or from exposure to something that causes health or economical losses. By a security property, we mean a property that guarantee of being protected from event or from exposure to something that causes availability, integrity or confidentiality losses.

An instance of a critical system could be the pitot tube and its associate electrical and software components. Pitot tubes are sensors that measure the speed of a flowing material (gaz or liquid). They are used in the aeronautic industry to give air-speed of a plane during a flight. If the pitot tube becomes deficient, the pilot could read false speed data and could have the wrong behaviour regarding the security of his passengers. Pitot tubes and wrong pilot's reactions were responsible of several plane crashes: the flight 301 Bir-

genair (1996) crashed probably because of a wasp hive inside the tube[5], the flight 2553 Austral Líneas Aéreas (1997)[6] and the flight 447 Air France (2009)[7] crashed because the pitot tube was caught in the ice while the aircraft was crossing a cloud.

To address complexity and criticality issues, complex and critical systems are generally built under strong constraints to obtain certifications. These kind of certifications guarantee the quality of components regarding security and safety properties. For instance the certification ISO 26262 [ISO, 2011] recommends test quality measurement over model coverage, code coverage and requirements coverage. More generally, the quality of a system is guarantee by **verification** and **validation** (V & V) activities. The following are some definitions from the ISO 9000 [ISO, 2005] normative document:

- Verification uses objective evidence to confirm that specified requirements have been met. It addresses the question of "Are we creating the **system right** ?".
- Validation uses objective evidence to confirm that the requirements, which define an intended use or application have been met. It addresses the question of "Are we creating the **right system** ?"

In other words, verification is the process of evaluating a system during a conception phase to determine whether it satisfies the conditions imposed at the start of that phase. Validation, on the other hand, is the process of evaluating a system at the end of a design process to determine whether it satisfies specified requirements. Verification and validation both focus on the fact that systems have to satisfy requirements. Requirements are grouped into two categories: functional and non-functional requirements. Functional requirements describe the functionalities that the system must be able to do. Non-functional requirements describe the quality of the services offered by the system including, for instance, security, reliability, performance, maintainability, etc.

During the verification step, engineers have to ensure that the model and its refinements satisfy the requirements for each use case. Some emerging techniques, based on formal methods, can be used to perform verification:

- Proving [Le Lann, 1998] enables to prove that some properties, defined by an user over a formal model, are always verified. It requires the use of interactive solvers, which need human expertise. Moreover, only the formal model is verified, not the real system.
- Model-checking [Clarke et al., 1999] enables to verify properties over a formal model by ensuring satisfiability on every reachable state of the system. This method requires to exhaustively get through the reachability graph described by or derived from the formal model.
- Simulation [Kleijnen, 1995], most often used in real-time systems development, also enables to verify the behaviour of a formal model and to check that some specified properties are satisfied. Several models must be developed to use simulation as a verification approach. Verification is performed by determining, with the use of simulation, whether the model of the system behaves as expected on models of the external environment.

[5] http://www.fss.aero/accident-reports/dvdfiles/DO/1996-02-06-DO.pdf [Last viewed on September 2015]
[6] http://aviation-safety.net/database/record.php?id=19971010-0 [Last viewed on September 2015]
[7] http://www.bea.aero/docspa/2009/f-cp090601/pdf/f-cp090601.pdf [Last viewed on September 2015]

While verification focuses on the model and its refinements, validation ensures that the real implementation satisfies the requirements for each use case. In the industry, simulation and testing are the most common activities to validate real-time systems [Krichen and Tripakis, 2009].

To sum up, we have pointed out that, although complexity may lead to an infinite number of states, decreasing risks and accidents requires the system to be as trusty as possible. The challenges raised by critical and complex systems are transferred to verification and validation issues since verification and validation activities enable to guarantee a certain degree of confidence.

For thirty years the verification and validation phases are part of development cycle [Adrion et al., 1982]. In an industrial context, design processes such as *In-the-Loop Processes* include verification and validation at the core of quality insurance. The next section introduces such processes.

1.1.1/ IN-THE-LOOP PROCESSES

Applying iterative and incremental approaches has helped the verification and validation of complex and critical embedded systems, especially within real-time domain. Such typical approaches, as depicted in Fig. 1.1, are known as *In-the-Loop* processes, and can be performed at different levels: Model-In-the-Loop (MIL), Hardware-In-the-Loop (HIL), Processor-In-the-Loop (PIL), and Software-In-the-Loop (SIL) [Broekman, 2002]. Simulation and testing are at the core of all these system design processes [Bullock et al., 2004, Grossmann et al., 2011]. For example, within MIL process, at the early stages of the design process, the system (or subpart of the system) and its environment (called the *plant* model) are modeled and simulated using languages such as Modelica[8] or Matlab-Simulink[9] to ensure that the designed (sub)system conforms to its requirements [Matinnejad et al., 2014]. This process is also known as rapid prototyping. Another level of simulation and testing concerns the HIL process and consists to test the real hardware platform in combination with its simulated environment [Benigni and Monti, 2011].

This process is also used when building control software that interacts with mechatronic systems. The controller software is designed and validated using SIL testing. The software is executed on an external processor board or is synthesized using a Hardware Description Language (HDL) and loaded into an FPGA (Field-Programmable Gate Array) board. At this point, I/O are simulated on external platforms (plant) and the implementation is validated against the controller design, model and requirements.

Since models are the entry point of these V&V activities, the next section introduces concepts about modelling of critical and complex systems.

1.1.2/ MODELLING OF COMPLEX AND CRITICAL SYSTEMS

Models are the common artifact to the previously described verification and validation techniques. A model is an approximation of selected aspects of the structure, behaviour, or other characteristics of a real-world system: a model is an abstraction. The level of abstraction is function of the model purpose.

[8] https://www.modelica.org/documents/ModelicaSpec33.pdf
[9] http://www.mathworks.fr/products/matlab/

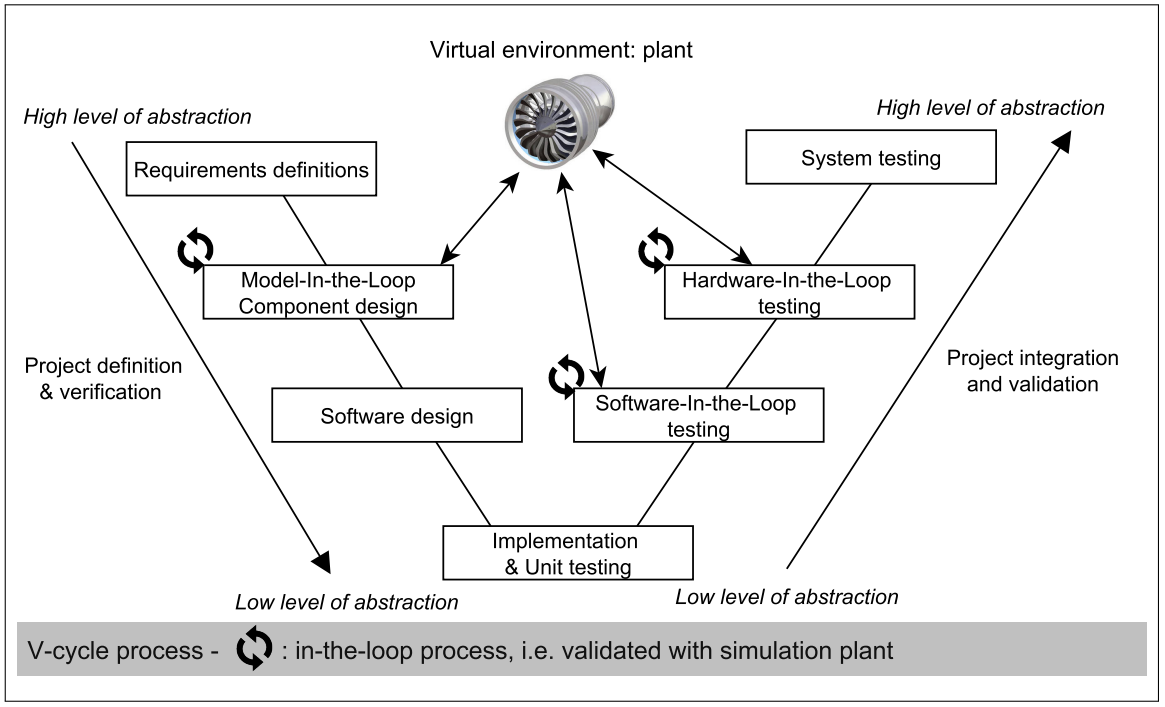


Figure 1.1: V-cycle Process using In-the-Loop Activities

For instance, in a classical black-box Model-Based Testing (MBT) [Pretschner, 2005] process, the model is abstract enough to become the specification. The model is formalized to enable test cases generation. The generated test cases are then used to validate the system (validation testing) against the requirements. A model for simulation purpose is less abstract than a model for MBT purpose. It generally contains equations and continuous features that enables to explore physical phenomena.

Whatever the level of abstraction of a model, it is first necessary to define its purpose and its scope. To define this scope, the notion of **separability** should be considered. This notion, also known as decomposition, simply implies that a system is defined by what composed its structure. What is called the perimeter of the investigation is defined by the fact that decomposition is limited in depth. It means that, at a certain degree of granularity, components are not decomposed anymore and are described as kind of gray box driven by physical laws in relation with the requirements. As a consequence the model of a system is bounded to its smaller specified elements. This defines the level of system specification.

The last important notion concerns the interactions between the system and its environments. Since separability defines a depth limit in the investigation, **selectivity** forces to limit the studying interactions. Modelling all possible interactions between a system and its environment is not feasible. Indeed, since models are abstractions of the reality, modelling assumes that most of the interactions of the system under study are ignored or simplified. In other words, what defined selectivity is the action of selecting only a small subset of possible interactions that are relevant to the purpose of the model. For instance, when modelling a mechanical system, we ignore acoustic, optical, and electrical interactions and consider only mechanical variables. Thus, the system is viewed as a separate entity that interacts with its environment through input and outputs.

The use of models to design a first overall system is increasing and models are becoming the starting point and the repository of new developments. To meet these expectations, the International Council On Systems Engineering (INCOSE)[10] has identified the Model-Based System Engineering (MBSE) as a key practice for efficient systems development. Indeed, since MBSE approach is replacing the traditional document centric approach, models are at the core of the requirements definition, design, analysis and verification/-validation activities [Wymore, 1993].

Basically, MBSE aims to achieve these activities using models that describe the system under development. This kind of approach is mostly supported using formal or semi-formal modelling artifacts, which are enough precise to achieve formal verification [Baracchi et al., 2013], but also simulation and testing that provide early practical feedback to validate requirements [Qamar et al., 2009]. Simulation code generation from formal model is increasing as it reduces the gap between high level of abstraction modelling and rapid prototyping, as demonstrated in [Sindico et al., 2011]. Finally, using formal model also enables to apply MBT approaches [Utting and Legeard, 2007] that aim to cross-check a model against an implementation, and hence make it possible to provide early validation of functional as well as non-functional properties, such as performance and resource use.

In this thesis, we focus on SysML as it becomes one of the most used modelling language in the MBSE field and as it meets the needs of engineers to describe all aspects of complex systems (from requirements to behavioral aspects). In addition, previous work on SysML have demonstrated its capability to perform MBT [Lasalle et al., 2011a]. However, SysML is not sufficient to study continuous physical behaviours and interactions of complex and critical systems.

Since the use of simulation is efficient to study physical phenomena, we introduce in the next section general assumptions about the simulation activity.

1.1.3/ SIMULATION OF COMPLEX AND CRITICAL SYSTEMS

In many cases, performing experiences on the real system is impracticable, too expensive or unethical. Then, the use of simulation enables to overcome these issues. It enables to explore phenomena using computers and to study and derive expected behaviours. A simulation can be achieved if engineers have enough acquired knowledge, obtained by previous experiments on similar phenomena.

Some simulations have a very high cost (still small compared with real experiments). This explains that industries, which use simulation, especially when using exceptional computing means, are the most value-added industries (aeronautics and space) or high risk industries (nuclear for instance). Indeed, simulation allows the implementation and validation of a large number of execution sequences in a limited time. Nevertheless, the development of simulated version can be time consuming.

The entry point of simulations is the model. Inputs, parameters and constraints are elements whose variations affect the behaviour of the model. We call a real-time simulation, a simulation that is performed at the same speed or faster than the actual phenomenon. It enables to perform digital-analog simulations that include humans as an analog element: this is called human-in-the-loop simulation [Chiang et al., 2010]. An aircraft simulator is a

[10] http://www.incose.org/ [Last viewed on September 2015]

good example: the pilot (analog) sits in a quasi-real cockpit (analog) and pilots his plane. The orders he gives are read by a computer that calculates the aircraft movements (digital). Actuators (analog), driven by the computer (or controller), allow the pilot to feel and see the effects of the orders he gave.

Similarly, in this case, the simulation loop may comprise real elements such as equipment and sub-systems to be tested. This is called *In-the-Loop* simulation. A good example is the testing of real ABS subsystem integrated in a driving simulator or managed by a software like SimulationX[11].

1.1.4/ TESTING OF COMPLEX AND CRITICAL SYSTEMS

According to G.Myers, testing is the activity of executing a system in order to find anomalies or defaults [Myers et al., 2011]. From the industry point of view, testing is often an empirical and manual approach. Testing is time expensive and it often ends when the delivery dates are reached (e.g. not when the system is considered safe). The success of a test campaign does not allow to conclude that the system is correct but only provides some confidence in the system. Indeed, testing does not allow us to assess that a system is faultless [Dijkstra, 1972]. However testing is a crucial step during the design of critical and complex systems. Testing activity is composed of four main steps:

1. **test conception**: from a model or from the specifications, the test engineer defines what has to be tested;
2. **test writing**: the test cases are written or generated to cover code structure or model elements;
3. **test execution**: the test cases are executed on the implementation
4. **test result evaluation**: the results of the test cases are compared to what is expected (oracle) to assign a verdict.

In addition, there are several test families:

- **unit testing**: verification of the functionality of a specific section of code at the function level;
- **integration testing**: verification of the interfaces and interactions between components against a system design;
- **regression testing**: validation of a system after a major evolution;
- **functional testing**: validation of a specific action regarding system's specification.

In the context of this thesis we focus on functional test conception and functional test generation. There are numerous test generation techniques. We decided to use model-based testing, which consists in generating test cases from an abstract model that represents the system under test. The system's behaviour is compared to the model's behaviour during tests execution.

[11] http://www.simulationx.com/

1.2/ SUPPORTED PROJECTS

The work presented in this thesis are incorporated within the framework of twenty years DISC research work on model-based testing, which foundation is the creation of a calculation program based on a solver dubbed CLPS (Constraint Logic Programming with Sets). In addition, we present in this thesis an approach initiated by the VETESS project (2008 to 2010). This previous work proposed a tooled approach that enabled to perform test generation from SysML models. Several weaknesses of this approach has been identified. To address the issues raised by the VETESS project, this thesis was supported by three projects, namely SyVAD, Smart Blocks, and GEOSEFA. Each of these projects is presented in this section. These projects brought together the following actors from the FEMTO-ST institute:

- The DISC department[12] (Département d'Informatique des Systèmes Complexes) is reputed for its expertise in automatic generation of test cases from abstract models.
- The ENERGY department[13] is specialized in the thematic of electrical energy, design of electrical and thermal machines, and fuel cell and hybrid vehicles. The involved researchers of the ENERGY department have guided our work according to their needs and they proposed the main case study of the project.
- The MN2S department[14] (Micro Nano Sciences et Systèmes) and the AS2M department[15] (Automatique et Systèmes Micro-Mécatroniques) were also involved (indirectly) in the project. The MN2S department is specialized in the micro and nanotechnologies. AS2M department's activities are based on the areas of automation, robotics and mechatronics.

1.2.1/ THE CLPS PROJECT (1990-...)

Since 1990, the DISC department is working on a solver dubbed CLPS [Legeard and Legros, 1991] for Constraint Logic Programing with Set. The first main contribution (1990 - 1993) of this project was to consider set structures with their associated operators ($=$, $\neq$, $\in$, $\notin$, $\cup$, $\subset$, $\cap$, $\forall$, $Card$) for logic programming. Then, from 1996 to 2004, several major contributions [Legeard and Py, 1999, Legeard et al., 2002, Legeard et al., 2004] to this solver were made to enable animation and test generation from B [Abrial, 1996] and Z specifications [Spivey, 1992a]. These work led to the development of the BZ-Testing-Tools [Ambert et al., 2002] (BZ-TT), which embedded the CLPS solver now called CLPS-BZ [Bouquet et al., 2004a].

BZ-TT has been industrialized by the Smartesting[16] company. BZ-TT is a tool-set for animation and test generation from B, Z and Statechart specifications. Each of these notation is translated into the intermediate format of the BZ-TT tool. This format is itself the input of a translation into a constraint system to enable symbolic evaluation for animation and test generation.

[12] http://www.femto-st.fr/en/Research-departments/DISC/Presentation/
[13] http://www.femto-st.fr/fr/Departements-de-recherche/ENERGIE/Presentation/
[14] http://www.femto-st.fr/en/Research-departments/MN2S/Presentation/
[15] http://www.femto-st.fr/en/Research-departments/AS2M/Presentation/
[16] http://www.smartesting.com/en/

Finally, the CLPS-BZ solver has been extended to support symbolic animation of object-oriented specifications with interacting objects, inheritance, dynamic object creation to animate JML [Leavens et al., 1998] specifications [Bouquet et al., 2005b, Bouquet et al., 2005a]. Since there were internal work to use CLPS-BZ as animation and test generation engine for UML specifications, and considering the long history of this solver, we decided to update and to extend it to animate SysML specifications and to generate test cases.

1.2.2/ FEEDBACK OF THE VETESS PROJECT (2008-2010)

The VETESS project (Verification of Embedded systems for vehicles using automatic TESt generation from Specifications) led to the development of a tooled process, which permits the automatic test generation from UML/SysML models. This process, illustrated in Fig. 1.2, permits to generate abstract test cases and to concretize them to be executed on a physical test bench or on a simulator.

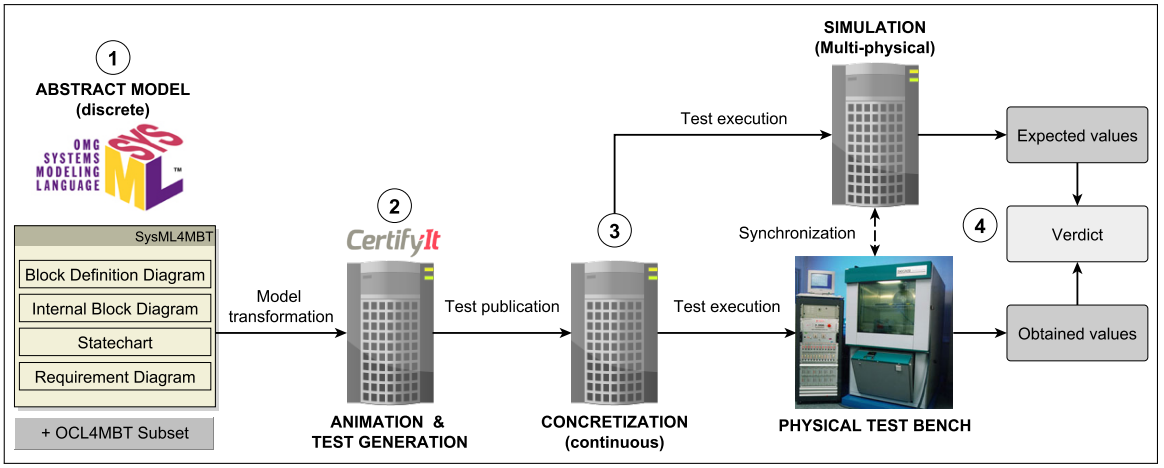


Figure 1.2: VETESS Model-based Tooled Process

The first step of this approach (①) consists to model the SUT (System Under Test) and its environment using a subset of the SysML language named SysML4MBT [Lasalle et al., 2011a]. This subset contains the Block Definition Diagram to represent the static view of the system, the Internal Block Diagram to represent the interactions between blocks, and a state diagram to represent the dynamic aspect of the system. Moreover, the requirement diagram permits to specify the requirements of the system. This ensure a bidirectional traceability between generated tests and initial requirements. Finally, the model is formalized by adding constraints expressed with a subset of the OCL [Warmer and Kleppe, 1996] language named OCL4MBT [Bouquet et al., 2007]. The SysML4MBT subset permits to discretize the SUT and its environment, which means that the model specifies the state of the system at specific times (reception of a stimulus) without studying variations between two stable states.

The second step (②) is the abstract test cases generation with the software Certify It^{TM} by applying structural coverage criteria (coverage of states, transitions, etc), and a def-use approach on signal exchange dubbed com-cover [Ambert et al., 2013]. These tests are abstracts as they are expressed with elements of the SysML4MBT subset.

The concretization step (③) consists of giving concrete values to the abstract elements of the generated abstract test cases [Lasalle et al., 2011b]. During this step, it is necessary to specify the time elapsing between two stimuli to obtain a test sequence, which can be executable on the real system. Therefore, continuous aspects are reintroduced at this level.

Finally (④), test sequences are executed on the real system and simulated using a Matlab framework, and obtained values are compared with values from simulation to assign a test verdict.

This approach is relevant to automatically generate many test cases while ensuring an optimal coverage of the model. Moreover, the feedback obtained during this project have been considered very satisfactory by PSA engineers who validated these results within the VETESS project [Ambert et al., 2012].

However, this approach has shown some weaknesses and the case studies permitted to identify several areas of improvement:

- the validation engineer has to specify both the SysML4MBT model and the MATLAB simulation model: the SysML4MBT model is based on a discretization of the real system, i.e. continuous aspects of the SUT are taken into account during the concretization step or in the adaptation layer. At this stage, the link between SysML4MBT and MATLAB models is explicitly realized,
- the adaptation layer must be corrected for each model modification, which makes maintenance very costly,
- the Certify It^{TM} test generator admits only one state machine (Cartesian product needed in case of several parallel state-machines to produce only one state-machine),
- the functional validation of the SysML4MBT model is checked during the execution of test cases in the simulation model: the late validation delays the development of the model and its stabilization can be time and effort consuming.

1.2.3/ THE SYVAD PROJECT (2011-2014)

The Regional SyVAD (SysML modelling approach for the Verification and the vAliDation of micro-systems) project was the main support of this thesis. It consists of defining and instantiating a tooled process to design, implement and validate physical and embedded systems. This project took place from 2011 October 1^{st} to 2014 September 30 and is based on the results obtained during the VETESS[17] project. It defines the tooled approach presented in this dissertation, based on the SysML formalism enabling 3 activities:

- the identification, formalization, and structuring of system requirements at the SysML model to ensure traceability,
- define transformation rules from SysML model to VHDL-AMS. However, after inconclusive experiments, we decided to opt for Modelica. Thus, we decided to get closer to the OMG SysML-Modelica working group. The OMG specification proposes to

[17] http://lifc.univ-fcomte.fr/vetess/

take into account the hierarchical description (hardware and / or software) of the system with blocks, and the description of the physical constraints. We considered the behaviour of each block using state-machines,

- define a method to generate test cases from the SysML model to serve as input to the Modelica simulation model.

1.2.4/ THE SMART BLOCKS PROJECT (2011-2014)

The French National Agency for Research (ANR) Smart Blocks project, which started in 2011, combines systems micro-technology, control theory and computer science to create self-reconfigurable modular conveyor based on contact free technology. Smart Blocks federates four French research laboratories and a Japanese one. This project enabled the following activities:

- the SysML modelling of the conveyor from specifications and mathematical equations,
- the simulation of the model in a Modelica simulation framework.

This case study enabled to assess the relevance of a SysML based simulation framework to validate requirements. In addition, the experiments permitted to validate the SysML to Modelica translation process.

1.2.5/ THE GEOSEFA PROJECT (2014 - 2017)

The Regional GEOSEFA project, which started in 2014, relates to energy management strategies for an electrical power system embedded in a new type of aircraft. The goal is the sizing and the optimization of the energy management with algorithms. Automated test generations are studied in order to validate the choices according to the constraints of the application.

In collaboration with the domain experts, the SysML model of the overall electrical power system has been designed. Then, Modelica simulations have been performed to calibrate (verification). Finally, the model of the environment (plant) has been used as input of automatic generation of test cases. This case study enabled to combine discrete and continuous modelling for simulation and testing activities. The obtained results were conclusive regarding the coverage and the number of generated scenario for validation activities.

1.2.6/ SYNTHESIS

This section has presented past and present projects. We have presented the BZ-TT tool and the CLPS-BZ solver has they are historically developed and used by the DISC department. Therefore, we have decided to use the CLPS-BZ solver to animate and generate test cases to validate the model and the real implementation of physical critical and complex systems. The CLPS-BZ solver is presented in detail in Chapter 3.

Then, we presented the VETESS project, which has led to new research questions. The main issues highlighted by this project concerns the late validation of the SysML model and the late consideration of the continuous aspects. To address these issues, the SyVAD project permitted to define a methodological simulation and testing approach based on SysML. As presented in Chapters 4 and 5, the proposed approach relies on the combination of continuous and discrete modelling.

Finally, the Smart Blocks project and in the GEOSEFA project enabled to evaluate the proposed approach with real-life case studies. The Smart Blocks case study, presented in the Chapter 7, permitted to perform early requirements validation, and thus to assess the relevance of using continuous Modeling within SysML. Then, the GEOSEFA case study, which is presented in the Chapter 8, enabled to assess the relevance of combining continuous modelling for simulation with discrete modelling for test generation.

The next section introduces and highlights the contributions of this dissertation.

1.3/ CONTRIBUTIONS OF THE THESIS

This work is motivated by the challenges raised during the design, the verification and the validation of critical and complex systems in a *In-the-Loop* context. In this section, we present the research questions and the contributions of the thesis.

1.3.1/ RESEARCH QUESTIONS

The VETESS project (see Sect. 1.2.2) highlighted areas for new research directions. From these needed improvements, the following research questions and sub-questions have been derived:

- **RQ1** In what extent is it possible to build a unified model for simulation and testing?
  - **RQ1.1** How to combine discrete and continuous domains in the same model?
  - **RQ1.2** How to enable simulation and test generation from the model?
  - **RQ1.3** How to verify and validate the model at the soonest?
- **RQ2** How to make executable such models?
  - **RQ2.1** How to convert the model into an interpreted and effective language?
  - **RQ2.2** Which solver is suitable for parallel state-machines animation?
  - **RQ2.3** Is there any way to use continuous solver (equation solving) in cooperation with discrete solver (CSP solver for instance)?
- **RQ3** In what extent the proposed approach could be automated?
  - **RQ3.1** From such a unified model, is it possible to automate simulation code generation in an effective manner?
  - **RQ3.2** In what extent the automatic test generation may be performed?
  - **RQ3.3** In what extent the test sequences execution and the verdict assignment may be automated?

These research questions led to the development of a tooled approach that enables to perform simulation and testing from a unique SysML model specifying the system under test and its environment. This tooled process is suitable for *In-the-Loop* processes as simulation and testing are at the core of such processes.

1.3.2/ CONTRIBUTIONS OVERVIEW

In the context of this thesis, we have focused our work on three aspects of the *In-the-Loop* process. Figure 1.3 simply illustrates the proposed model-based validation approach for complex systems. First, we propose to use SysML to model and specify a real-time system and its associated environment (virtual plant). The use of SysML is motivated by two reasons: SysML is abstract enough (close to requirements) and is efficient to be the entry point of simulation and testing activities.

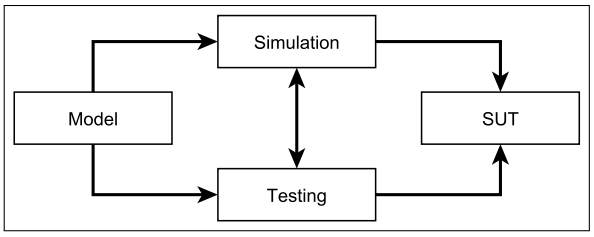


Figure 1.3: Model-based Validation Process for Complex Systems

Second, we propose an approach that enables to perform verification over the SysML model using Modelica simulation. This approach defines a contribution to the MIL (or rapid prototyping) process by enabling automatic generation of Modelica prototypes and their associated Modelica virtual plant using Model Driven Engineering techniques.

Third, we propose to validate the SysML model of the system and its implementation by generating test cases from the model of the virtual plant. This defines a contribution to MIL-testing and HIL-testing.

This thesis describes an original and innovative SysML-based formal framework for simulation and testing of multi-physical and critical systems, that bridges the gap between high-level design model, starting point of MBSE approaches, and real-time execution platform, keystone of the In-the-Loop approaches. In this way, this framework allows system engineers to stay as close as possible of the initial design specifications when achieving all the steps of the development life-cycle. Moreover, it takes advantage of both approaches by ensuring a model centric process enabling validation, simulation and testing from the earliest stage of design. To achieve that, the architecture and the discrete behaviour of the system are described by a SysML model, which is annotated with OCL and Modelica code to specify its discrete and continuous features. This model is used to automatically generate real-time Modelica program for simulation, and black-box test cases for validation. The generated test cases can be simulated using the generated Modelica program to validate the design model as well as the physical system itself. Therefore, the proposed framework can contribute both to MIL process (model against simulated environment), and to HIL process (physical system against simulated environment). The implementation of the whole process allowed to assess the relevance of the approach in an operational context.

1.4/ THESIS OVERVIEW

This dissertation is divided into fours main parts. Part I situates the work presented in this thesis in the existing scientific context. In the part II we detail the contributions of this thesis, in particular the combination of discrete and continuous features in SysML for model-based testing and simulation purpose. Part III presents the experiments and the obtained results. Finally, we conclude and outline future work in part IV.

Part I Context, Motivations and State of the Art
Chapter 1 has introduced the context and the motivations of the thesis. In addition, we will present the running example at the end of this chapter.
Chapter 2 introduces related work about modelling, simulation and testing of complex and critical systems.

Part II Contributions: Discrete and continuous modelling in SysML
Chapter 3 presents the proposed approach of simulation and test generation from SysML models.
Chapter 4 presents the integration of Modelica constructs into SysML models and the process of Modelica code generation from such models.
Chapter 5 proposes a translation of SysML models into CSP (Constraint Satisfaction Problem) to enable animation and test data generation.

Part III Development and Experiments
Chapter 6 presents the implementation of the elements presented in the previous sections (it defines a proof of concept). This chapter describes the technologies and connectors implemented for the definition of an automated tooled process.
Chapter 7 presents and discusses experiments and obtained results on the SmartBlocks case study.
Chapter 8 presents the application of the overall approach, experiments and results over a case about a new electrical power system (EPS) for an aircraft.

The last part of this dissertation concludes our work and presents potential improvement of the proposed SysML-based framework and process for simulation and testing.

1.5/ RUNNING EXAMPLE: TANK SYSTEM

This section introduces example of a tank system inspired from [Fritzson, 2010]. This classical example is used to illustrate every concepts described in this thesis. As depicted in Fig. 1.4, we consider two tanks serially connected with single inlet and outlet streams. Each tank is connected with a controller that controls the valve opening depending on the liquid height. In this running example, we consider the environment as the possible source of liquid, i.e the rain or tap of water for instance.

1.5.1/ MATHEMATICAL MODEL OF THE TANK

The following is the mathematical model of the tank. The equation (1) describes the variation of the tank's liquid height h depending on the inlet f_{in} flow rate, the outlet f_{out} flow rate, and on the tank's base area s. The sensor of the tank measures the height of liquid, as described by equation (2).

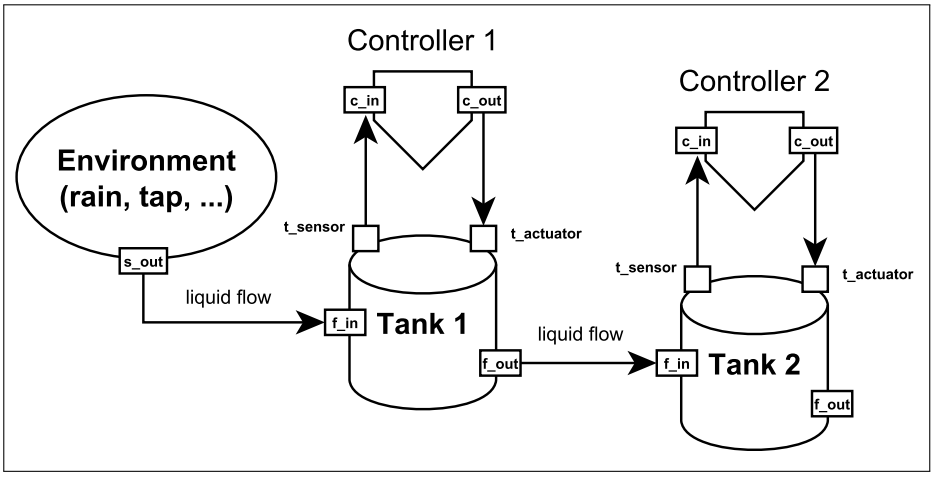


Figure 1.4: Tank System Overview

$$\dot{h} = \frac{f_{\text{in}} - f_{\text{out}}}{s} \tag{1}$$

$$t_{sensor} = h \tag{2}$$

h: tank level (m), f_{in}, f_{out}: flow rate ($m^3 \cdot s^{-1}$), s: area (m^2).

As described by equations (3), (4), and (5), the output flow f_{out} is related to the valve position by a flow gain parameter $flowGain = 0.02\ m^2 \cdot s^{-1}$, and by a limiter that guarantees that the flow does not exceed what corresponds to the open/close positions of the valve. Thus, $minV = 0$ is the lower limit for the output valve flow, and $maxV = 10$ the higher limit for the output valve flow. The tank's actuator measures is controlled by the controller (6).

$$-flowGain \cdot t_{actuator} > maxV \implies f_{out} = maxV \tag{3}$$

$$-flowGain \cdot t_{actuator} < minV \implies f_{out} = minV \tag{4}$$

$$minV < -K \cdot t_{actuator} < maxV \implies f_{out} = -flowGain \cdot t_{actuator} \tag{5}$$

$$t_{actuator} = c_{out} \tag{6}$$

1.5.2/ MATHEMATICAL MODEL OF THE CONTROLLER

The controller need to be specified. We will choose a PI (proportional and integrating) controller. Such controller is used to reach a desired value. For example, the desired value may represent the location we want a robot to move to, the speed we require an engine to reach, or the position we need a valve to arrive at. In our running example,

the desired value represent the liquid height in a tank. Such controllers have to satisfy a number of requirements:

Liveness (functional): The controller shall guarantee that the actual value will reach the desired value within $t1$ seconds. This ensures that the controller indeed satisfies its main functional requirement.

Stability (safety, functional): The actual value shall stabilize at the desired value after $t2$ seconds. This is to make sure that the actual value does not divert from the desired value or does not keep oscillating around the desired value after reaching it.

Smoothness (safety): The actual value shall not change abruptly when it is close to the desired one. This is to ensure that the controller does not damage any physical devices by sharply changing the actual value when the error is small.

Responsiveness (performance): The difference between the actual and desired values shall be at most $v3$ within $t3$ seconds, ensuring the controller responds within a time limit.

The behavior of a continuous PI controller is primarily defined by the equations (7), (8) and (9):

$$\text{delta} = ref - c_{\text{in}} \tag{7}$$

$$\dot{x} = \frac{delta}{T} \tag{8}$$

$$c_{out} = K \cdot (delta + x) \tag{9}$$

Here x is the controller state variable, $delta$ is the difference between the desired reference level ref and the actual level of liquid obtained from the sensor. Discrete controllers repeatedly sample the fluid level and produce a control signal that changes value at discrete points in time with a periodicity of $T = 10$ seconds. Finally, c_{out} is the control signal to the actuator for controlling the valve position, and K is the gain factor.

1.5.3/ ENVIRONMENT MODELLING

In this running example, we consider the source of liquid as the environment of the system. Therefore, we have a liquid source component. The flow increases sharply at $time = 150\ s$ to factor of three of the previous flow level, which creates a interesting control problem that the controller has to handle (equations (10) and (11)).

$$time < 150 \implies s_{out} = 0.02(m^3 \cdot s^{-1}) \tag{10}$$

$$time \geq 150 \implies s_{out} = 0.06(m^3 \cdot s^{-1}) \tag{11}$$

2

STATE OF THE ART

Contents

This chapter presents the current state of scientific work and languages related to the modelling, simulation and model-based testing for the validation of physical systems. Consequently, the state of the art is divided into three sections. After introducing modelling languages in Sect. 2.1, we give details about simulation languages that are used in the industry in Sect.2.2. Finally, model-based testing methods, that exist both in the research field and in the industry, are explored in Sect. 2.3.

2.1/ HIGH-LEVEL MODELLING OF COMPLEX SYSTEMS

In this section, we present a state of the art of modelling languages that exist in the industry and in the scientific literature. To stay close to the system engineering business knowledge and thus to facilitate the evaluation and the adoption of our work by professionals in this field, we choose to focus on graphical and standardized languages that enable to model with a high level of abstraction structural and behavioral part of a physical system.

The evaluation criteria, and related questions are:

- **EC1** Language scope:
  - **EC1.1** Does it support structural and behavioural modelling?
  - **EC1.2** Does it support multi-disciplinary modelling?
  - **EC1.3** Does it support equation writing or modelling?
- **EC2** Level(s) of abstraction:
  - **EC2.1** Does it support high-level modelling?
  - **EC2.2** Does it support requirements management and traceability?
- **EC3** Language confidence:
  - **EC3.1** Is the language standardized or normalized?
  - **EC3.2** Is the language used by an active community?

2.1.1/ THE HILES LANGUAGE

Hiles is a formalism, which was founded by the LAAS-CNRS in 2000. Hiles is an extension to the Petri nets that is used to describe and specify the behavior and structure of systems at a high level of abstraction. Methodologically, Hiles applies a top-down approach in accordance with the principles of systems engineering. The work of [Goómez et al., 2010] helped to develop a platform, named Hiles Designer for the modelling and generation of VHSIC Hardware Description Language - Analog and Mixed Systems (VHDL-AMS) [Peterson et al., 2002]. The Hiles Designer platform was used during the earlier stage of design, in communication with TINA [Berthomieu and Vernadat, 2006] and VHDL-AMS simulators.

The main objectives of Hiles are:

- The functional decomposition of the system: identify the main functions of the system and determining how they should be connected.
- The prioritization and architectural restructuring: allows a hierarchical and structural decomposition of the basic functions of the system using building blocks. The structural blocks can contain other building blocks, functional blocks to describe the system in the form of Differential and Algebraic Equations (DAEs). Finally a structural block may also contain a hierarchical control network based on Petri nets.
- Building a formal representation of the sequence of operations. The developed models enable formal verification: Petri nets provide a fairly comprehensive and powerful model for managing parallelism and concurrency.
- Compatibility with a standardized language for multi-physical systems.

Regarding the criteria, Hiles is a high-level modelling language that supports structural, behavioural, multi-disciplinary, and equation modelling. However, it does not support requirements management and Hiles Designer is an academic tool that has not really found its place in the industry.

Finally, there is still a lot of experimentation to define the issues of multi-physical simulation, especially with the use of Unified Modeling Language (UML) and SysML that would establish a common methodology for modelling multi-physical systems and the generation of simulation code. The following section then presents the UML language.

2.1.2/ THE UML LANGUAGE

UML [Fowler, 2004] is an object-oriented modelling language initially designed for software modelling. UML is derived from a consensus between three object-oriented methods namely Booch [White, 1994], OMT [Frost, 1994] and OOSE [Jacobson et al., 1994]. UML is now specified by the Object Management Group (OMG). UML defines 14 diagrams that are hierarchically dependent to allow the modelling of a project throughout its life cycle. Although the UML semantic is not suitable for multi-physical systems modelling, there are many work on physical modelling and simulation code generation from UML diagrams.

STRUCTURAL MODELLING

UML provides three diagrams for the structural modelling of software. First, the UML class diagram allows engineers to model static structure of a system. Then, the object diagram is used to instantiate abstract objects defined in the class diagram. For instance, initial state of a system can be illustrated with an object diagram. Finally, the component diagram enables to specify required and provided interfaces between components. The component diagram is used to describe systems with Service-Oriented Architecture (SOA).

STATE MACHINE

A finite state machine is defined by an automaton composed of states and transitions. State machines are a kind of Labelled Transition Systems (LTS) defined by the tuple $LTS =< S, s_0, \Sigma, \delta >$ where S is a set of states (potentially infinite), s_0 is the initial state, Σ is the alphabet of labels and δ is the transition relation between states defined as $\delta : S \times \Sigma \rightarrow S$. The UML state machine semantic has been defined in collaboration with Harel [Harel, 2007] (father of state charts [Harel, 1987]).

Regarding the criteria, UML is a high-level modelling language that supports structural and behavioural modelling. However, UML is mainly used for software development: it does not support multi-disciplinary, equation and requirements modelling. Since UML is normalized and is used by an active community, there is now a UML profile called SysML, with a semantic adapted to the modelling of multi-physical systems. The next section presents then the SysML language and existing work on the use of SysML through the automatic generation of simulation code.

2.1.3/ THE SYSML LANGUAGE

To support MBSE principles, the OMG is developing SysML since 2001. This language enables system engineers to specify all aspects of a complex system using graphical constructs. SysML is built on the well-known UML by bringing adapted semantics to the system engineering field: SysML is a UML profile. For instance, SysML extends UML *Classes* into *Blocks* and it extends UML *Components* with *Ports* into *Parts* with *FlowPorts*. Finally, SysML provides two new diagrams that enable requirements modelling and parametric modelling. In summary, SysML is based on four pillars to represent four main diagram types: structure diagrams, behavior diagrams (interaction, state machine, activity), requirements diagram and parametric diagram.

Regarding the criteria, SysML is a high-level modelling language that supports structural, behavioural and multi-disciplinary modelling. Moreover, it supports equations modelling with the parametric diagram (or equation writing with constraints), and requirements modelling with the requirements diagram. Finally, SysML is normalized by the OMG and is used by an active community both in research and in the industry [Braunstein et al., 2014]. For instance, the work of Jarraya et al. [Jarraya et al., 2007] propose a new approach for automatic verification and analysis of the performance of the activity diagrams SysML. The issue is about the importance of time in modelling and analysis of real-time systems. Indeed, behaviours of real-time systems take time to execute and finish.

Complex and critical systems are defined by continuous components that interact with the environment. The time is therefore an important dimension of systems. Thus, we present languages that enable to specify time constraints and events.

2.1.4/ TIME MODELLING

First of all, we can cite the TEmporal Property Expression (TEPE) language [Knorreck et al., 2011]. This language extends the expressiveness of property language with the notion of physical time and unordered signal reception. It graphically enriches the SysML parametric diagram for describing logical and temporal properties. TEPE is then included in the AVATAR [Pedroza et al., 2011] SysML profile for performing formal verification in UPPAAL.

Timed UML and RT-LOTOS Environment (TURTLE) [Apvrille et al., 2004] adds a formal semantics for UML dedicated to the expression of real-time. UML class diagram and activities diagram are extended with composition and temporal operators. The formal semantic is given in terms of RT-LOTOS. TURTLE is also included in the AVATAR SysML profile.

UPPAAL [Larsen et al., 1997] is an integrated tool that enables the modelling and verification of real-time systems where timing aspects are critical (communication protocols or real-time controllers for instance). With UPPAAL, systems can be modeled as a of non-deterministic processes communicating through channels or shared variables [Yi et al., 1994]

Regarding SysML, the elements for modelling the effects of time flowing are the parametric diagram (modelling of mathematical equations) and the sequence diagrams (description of the execution of actions in time). In addition to these native structures, there are languages or annotations dedicated for expressing real-time constraints such as TURTLE or MARTE UML profiles.

Modeling and Analysis of Real Time and Embedded systems (MARTE) [OMG, 2011] is a UML profile specified by the OMG. It defines stereotypes that permit to model clocks and events. Using UML and MARTE for embedded-systems modelling is efficient [Iqbal et al., 2012b]. Finally, MARTE provides complementary elements with SysML [Espinoza et al., 2009].

2.1.5/ OTHER LANGUAGES

The following modelling languages are cited to give an overview of the first work in the modelling field for verification purpose. Some of them (contract-based modelling, LTS) are at the basis of recent modelling methodologies.

KRIPKE STRUCTURE

A Kripke structure [Kripke, 1959] is a calculus model, similar to finite state automaton, invented by Saul Kripke. Kripke structures were initially used to perform properties verification, specified with temporal logic, using model-checking techniques [Clarke and Emerson, 1982]. Each state is labeled with a set of logical propositions. When a given property is violated by the automaton, it is possible to get the state sequence that violate the property.

CONTRACT BASED MODELLING

There is a wide range of languages based on contract modelling techniques. Such model formally specifies the conditions that each operation of the system has to satisfy (precondition) and the service that the operation agrees to provide (post-condition). This type of language are not graphical but can produce textual specifications based on a formal notation. The most known languages are Eiffel [Meyer, 1988], VDM [Borba and Meira, 1993], Z [Spivey, 1992b], B [Abrial, 1996], JML [Leavens et al., 2006], and OCL [Warmer and Kleppe, 1996].

Because of their formal nature, these languages are used both during verification and validation. For instance, the LTG-B [Bouquet et al., 2004b] tool offers automatic test generation based on decision and data coverage from a B specification. The JML language is also used in the field of JAVA applications by test generation [Bouquet et al., 2006]. Within this same paradigm, OCL is often associated with UML models to formally specify behavior using constraints as preconditions and postconditions.

LABELED TRANSITION SYSTEM

A LTS (Labeled Transition System) is defined by a tuple $LTS =< S, s_0, \Sigma, \delta >$ as defined in 2.1.2. Examples of modelling language based on such semantics are Π-calcul [Milner, 1999], LOTOS [Logrippo et al., 1992], SDL [Belina and Hogrefe, 1989], and Promela [Holzmann, 1993, Holzmann, 1997].

This kind of language can be extended with, for instance, the definition of Input / Output: this is called IO-LTS (Input/Output Labelled Transition Systems). Inspired by the IO-LTS, a more abstract specification language was created: IOSTS (Input / Output Symbolic Transition Systems). These two languages are well suited for test generation and have been the basis of many tools such as TGV [Jard and Jeron, 2005] and STG [Clarke et al., 2002].

LUSTRE AND ALTARICA

For critical systems, we can cite the SCADE framework[1] (Safety Critical Application Development Environment) based on the Lustre language [Halbwachs, 2005]. Lustre is a synchrone and declarative language used for the design of critical softwares for the aeronautic, railway, and nuclear industry.

Finally, the Altarica language [Griffault et al., 1998] is a high-level modelling language based on hierarchy of nodes (components). It enables to check safety properties using model-checking techniques.

SYNTHESIS

Kripke structures and LTS are suitable for verification process because they offer a simple and formal notation. This formalism allows the expression of strict properties but lacks of semantic for structural constructs, such as decomposition and component definition. The LTS has the advantage and the disadvantage of potentially having an infinite number of states.

[1] http://www.esterel-technologies.com/

Table 2.1: Modelling Language Evaluation Criteria Matrix

	EC1 Scope			**EC2** Abstraction		**EC3** Confidence	
Language	EC1.1	EC1.2	EC1.3	EC2.1	EC2.2	EC3.1	EC3.2
HILES	✓	✓	✓	✓	✗	✗	✗
UML	✓	✗	✗	✓	✗	✓	✓
SysML	✓	✓	✓	✓	✓	✓	✓
AVATAR	✓	✓	✓	✓	✓	✗	✗
MARTE	✓	✓	✗	✓	✗	✓	✓

The finite state machines, such as state machine diagrams, permit to represent finite automata with more details than Kripke structures. The diversity of manipulable entities and the presence of high level complex structures (composite states ...) enable to assign specific semantics to modeled elements. The finite state machines are usually represented graphically, which can be an advantage in an industrial context.

The disadvantage of this type of language is the lack of semantics that can however be overcome by the use of contract such as OCL. These languages allow the representation of preconditions and postconditions to precisely formalize the characteristics of a given behavior. Because of their formalism and their comprehensive enough expressiveness, these languages are used to develop efficient automated processes.

2.1.6/ MODELLING LANGUAGES ASSESSMENT

In this dissertation, we focus on simulation and test generation from high-level models. The need of high-level models is motivated by the idea that requirements have to be always satisfied. High-level models define links between informal specifications (subject to interpretations) and the first implementation of the system (simulation). In an MBSE context, high-level models become the specifications.

Although all languages in this section allow the representation of an embedded system's behavior, some are more suitable than others (see Table 2.1), given the tooled process we propose. The initial model has to be expressed in an enough comprehensive language to specify correctly the structure, the behavior of a system, while having requirements management and traceability capabilities.

In this thesis, we have chosen SysML to be the high-level modelling language in a *In-the-Loop* context. It enables to specify the structure, the behavior and the requirements of embedded systems. Moreover, this language has been adopted by the MBSE community to be the starting point of new developments [Rashid et al., 2015a, Rashid et al., 2015b]. Hence, we propose to use SysML as the main artifact for the design, simulation, and test generation process. The OCL language was selected to formally specify component's behaviour within SysML state machine diagrams. Specifying time constraints is an important step during the design of embedded systems. Thus, we have planned to use MARTE, especially the *TIME package*, but we have not addressed this issue in this thesis.

Finally, SysML is not executable: it does not include an action language and there is no solid simulation framework to simulate SysML models with equations. However, simulation is known to be an efficient way to validate models (preventing from modelling errors) and to eliminate bad design choices. In the next section, we present the languages related to the continuous simulation of physical systems.

2.2/ SIMULATION OF COMPLEX SYSTEMS

Simulation is the main activity during the conception of a physical complex system. This section presents an overview of simulation languages for physical systems. The chosen language will be the target language of model transformations from SysML models. In order to choose the right language, we have to defined several criteria that are important to address issues presented in the introduction. The following are the criteria that the simulation language has to respect:

- **EC1** Language scope:
  - **EC1.1** Does it support the simulation of multi-domain systems (electrical, mechanical, fluidic, thermal, ...)?
  - **EC1.2** Does it support Differential and Algebraic Equations (DAE's) modelling?
  - **EC1.3** Does it support continuous and discrete modelling?
- **EC2** Modularity:
  - **EC2.1** Is there a common interpretive semantic between SysML and the selected simulation language?
  - **EC2.2** Does it support components decomposition and reuse?
- **EC3** Language confidence:
  - **EC3.1** Is the language standardized or normalized?
  - **EC3.2** Is the language used by an active community?
  - **EC3.3** Is a simulation environment (preferably free and open-source) available to conduct experiments?

For each languages, we present their objectives and their specificities regarding the criteria previously defined. First of all, the Hardware Description Language (HDL) VHDL-AMS and System-C (AMS) are presented since these languages are common in the embedded systems industry. Then, we focus on the Matlab-Simulink simulation language, and on the Modelica Language because they enable to describe continuous behaviours in an equation-based and object-oriented paradigm.

2.2.1/ HARDWARE DESCRIPTION LANGUAGES

In this section, we introduce the VHDL-AMS and System-C Hardware Description Languages. These languages are massively used in the industry to model embedded systems. We also present related work concerning UML / SysML models translation to such languages.

VHDL-AMS

VHDL-AMS is a hardware description language whose role is to check the behavior of a system composed of an analog part and of a digital part. VHDL-AMS has been normalized in December 1999 under the IEEE 1076.1-1999 reference. It has been developed as an extension to the VHDL language which does not permit to represent continuous time. The AMS part of the language enables to represent analog and mixed signals in a continuous fashion. Moreover, these signals supports discontinuities in time and can be described by differential and algebraic equations. Therefore, these signals permit to simulate physical events as speed, mechanical force, pressure and fluidic flow, among other, thanks to the generalized application of the Kirchhoff laws (energy conservation).

Carr et al. [Carr et al., 2004] propose to use the class diagram to generate VHDL-AMS code. The semantic is adapted: associations do not represent attribute typed with class but inputs, outputs and connectors between classes. This is debatable since the UML component diagram owns the coupled model semantic.

In [Rieder et al., 2006], the authors propose to transform UML instances into VHDL component. The transformation of UML instances is done in two steps: first, a VHDL component is defined within a VHDL architecture that contains the instances. The second step is to create an instance of the component and to connect them using the links of the object diagram as connectors.

Finally, the work of [McUmber and Cheng, 1999] propose matching semantics between UML state machine and VHDL. Moreover, the work of [Wood et al., 2008] bring the concept of model transformation to generate VHDL from UML state machines.

Regarding the defined criteria, VHDL-AMS enables simulation of multi-physical systems while supporting DAE's modelling and continuous and discrete paradigms. Moreover, it supports components decomposition and reuse and is used by an active community. We have first chosen this language to be the target of code generation from SysML models. We have done some experiments to generate VHDL-AMS code from SysML diagrams [Bouquet et al., 2012] based on the work of RTaW [2]. Despite that VHDL-AMS code can be easily generated by MDE techniques, we have faced to a major issue: there are many important instructions (`generic map` for instance) that have not obvious matching with SysML constructs. Moreover, VHDL-AMS does not provide convenient open-source simulation platform.

SystemC

SystemC is a HDL derived from C++ for modelling hardware systems as well as software or hardware-software systems (co-design). It was developed by several companies (ARM Ltd, Cadence Design Systems, CoWare, Mentor Graphics, and so on) and has been standardized by the IEEE in 2005 [IEEE, 2005].

SystemC is based on a set of C++ libraries for software/hardware modelling through executable specifications at multiple levels of abstraction. It enables to model specific OS entities such as semaphores or abstract communication channels (Ethernet, UMTS for instance).

[2] http://www.realtimeatwork.com/ [Last visited on February 2015]

There are several works concerning SystemC [3] code generation capabilities from UML and SysML models. The originality of the approach in [Peñil et al., 2010] is the automatic generation of SystemC code from generic MARTE models. In [Boutekkouk, 2010], full SystemC code is generated from UML sequence diagram and from UML activity diagrams whose actions are expressed with C++ Action Language. To conclude with SystemC code generation, we can cite [Raslan and Sameh, 2007] whose purpose is to accelerate System on Chip (SoC) design process by providing automatic translation of SysML designs into SystemC models.

Concerning the criteria, SystemC is well suited for the design of SoC but in our context it does not provide any solution to describe DAE's. However, an extension to SystemC, named SystemC-AMS [Farooq et al., 2010], has been developed for analog and continuous signal modelling. Despite it has been proved feasible to generate SystemC (AMS) code from UML and SysML models, the community of SystemC (AMS) is still confined to SoC engineers in the domain of wireless sensors, integrated circuits, etc.

2.2.2/ MATLAB - SIMULINK

MatLab is a framework of numerical simulation created by the MathWorks [4] company. It enables the manipulation of matrices, functions and data plotting, algorithms implementation, and it interfaces easily with other programs.

MatLab can be completed with multiple tools, called plugin or toolbox:

- Simulink: graphical simulation framework,
- Control System Toolbox: analysis of linear time-invariant models,
- Neural Network Toolbox: neural networks modelling,
- Optimization Toolbox: problem optimization,
- Statistics Toolbox: statistical models modelling and analysis,

Simulink is a graphical extension of Matlab for creating diagrams using blocks. These diagrams are used to represent systems and mathematical functions. Vanderperren Y. et al [Vanderperren and Dehaene, 2006] discussed two possible integration methods between UML or SysML models and MATLAB-Simulink. In [Sjöstedt et al., 2007], a procedure for transforming Simulink models to UML composite structure and activity models is presented. The transformation has been implemented using ATL. Other works [Kawahara et al., 2009, Qamar et al., 2009, Sindico et al., 2011] concern the transformation of SysML models to Matlab-Simulink model and are compliant with OMG Model Driven Architecture (MDA). MatLab is very used in the industry. It fits completely the needed criteria despite the fact that there is no open-source simulation framework for MatLab models.

[3] http://www.accellera.org/downloads/standards/systemc [Last visited in February 2015]
[4] http://www.mathworks.fr/ [Last visited on February 2015]

2.2.3/ THE MODELICA LANGUAGE

Modelica [Association, 2012] is a non-proprietary, object-oriented and equation based language adapted to complex physical systems modelling [5]. Modelica is built on acausal modelling with mathematical equations and object-oriented constructs, and is designed to support effective library development and model exchange. Finally, OpenModelica [6] offers a free and powerful simulation engine to do practical experiments and validate the proposed approach.

A mapping between SysML and Modelica has been proposed by Vasaiely [Vasaiely, 2009]. An interesting mapping between SysML parametric diagrams and Modelica equations is proposed. Moreover, this work is one of the OMG SysML-Modelica specification [OMGSM, 2012] initiator.

In [Pop et al., 2007, Schamai et al., 2009], the authors propose to apply the ModelicaML profile on UML models and to generate Modelica code directly with the Acceleo environment. A powerful Eclipse plugin for the Papyrus modeler is available but these works are not based on the OMG SysML-Modelica specification and do not take into account SysML models.

Schramm and al. [Ji et al., 2011] introduce the MDRE4BR profile (Model Driven Requirements Engineering for Bosch Rexroth) which aims to perform verification of the design against the requirements using executable model. This profile extends the current SysML requirements constructs and is linked with ModelicaML to transform analytical models into executable Modelica models.

Nytsch-Geusen [Nytsch-Geusen, 2007] also proposes to use a special forming of UML named UMLH, for the modelling of hybrid systems. Modelica code can be obtain automatically from UMLH models. However, the generated code has to be filled up with the physical equations of the system.

Several works have been done on Modelica integration in SysML models. The representation of Modelica models in SysML was first introduced by Johnson et al. [Johnson et al., 2012]. This work explores the definition of continuous dynamics models in SysML and the use of triple graph grammar to maintain a bidirectional mapping between SysML and Modelica constructs.

Finally, a java based implementation of the OMG SysML-Modelica Transformation using MagicDraw SysML has been proposed in [Reichwein et al., 2012] but it does not use model transformation and code generation technology.

Regarding the criteria, the Modelica language appears to be a good candidate. It supports equation modelling, continuous modelling, and discrete modelling for multi-physical systems. In addition, the related work shows that it's object-oriented paradigm and its semantic is close to SysML. Finally, this language is used by an active community and the OpenModelica environment is open-source and enables to perform simulations and interactive simulations.

[5] Commercial and free simulation environments: CATIA, Dymola, MapleSim, OpenModelica, etc.
[6] https://www.openmodelica.org/ [Last visited in June 2015]

Table 2.2: Simulation Language Evaluation Criteria Matrix

Language	**EC1** Scope			**EC2** Modularity		**EC3** Confidence		
	EC1.1	EC1.2	EC1.3	EC2.1	EC2.2	EC3.1	EC3.2	EC3.3
VHDL	✗	✗	✗	✓	✓	✓	✓	✗
VHDL-AMS	✓	✓	✓	✓	✓	✓	✓	✗
System-C	✗	✗	✗	✓	✓	✓	✓	✗
System-C AMS	✓	✓	✓	✓	✓	✓	✓	✗
Matlab	✓	✓	✓	✓	✓	✓	✓	✗
Modelica	✓	✓	✓	✓	✓	✓	✓	✓

2.2.4/ SIMULATION LANGUAGES ASSESSMENT

In the previous section we have explored a number of simulation languages. Some of them are well suited for our need of simulation and model transformation from SysML model. The Table 2.2 summarizes the characteristics of each simulation language regarding the evaluation criteria. For instance, MatLab - Simulink would be a good candidate. However, considering the number of involved partners we have decided to head for the Modelica language. It supports the simulation of multi-physical systems from a continuous and discrete point of view. Moreover, Modelica is used by an active community and it exists open-source simulation environment (OpenModelica) to do experiments. Finally, the OMG promotes a dedicated standard (SysML-Modelica Transformation standard) to integrate Modelica semantics into SysML. Thus, we propose to perform model transformation and code generation from high-level SysML models to the Modelica language. For more precision about tools and simulation languages, we refer the interested reader to a more detailed study, which can be found in [Carloni et al., 2006].

We want to propose a tooled process that enables to perform simulation and model-based testing from a unique SysML model. Then, the next section introduces concepts, practices, and related work about testing of critical and complex systems.

2.3/ TESTING OF COMPLEX SYSTEMS

The industrial practice for functional testing is mainly manual and empirical. The limits of these practices are being stretched by the increase in complexity of the systems being validated. Another difficult challenge is posed by the need to keep the pace with continuously evolving requirements, by which also testing is more and more to be seen as a continuous activity along the life cycle.

The basics of testing rely on back-to-back testing. The goal of back-to-back testing [Vouk, 1990] is to determine that an implementation and model both produce the same outputs when given the same inputs. There are four essential requirements for back-to-back testing to be successful. First, there should be a high degree of confidence in the correctness of the model due to prior testing of the model against its requirements. Second, the implementation should produce outputs which are reasonably close (small differences are likely to rounding of results during numerical calculations) to the outputs of the model for all inputs. Third, the tests which were used to perform the comparison should achieve a high degree of coverage of the model and its requirements. Fourth, the

tests should achieve a high degree of coverage of the implementation. This last point deserves some emphasis because, if we simply take the tests which were used during model verification and apply them to the implementation, we may satisfy the first three criteria without covering the full implementation structure. All the testing techniques presented below are part of back-to-back testing.

2.3.1/ MODEL-BASED TESTING

Model-based testing [Utting and Legeard, 2007] refers to the automatic generation of tests from a precise model of the system. Various modelling notations, such as state charts, UML [Bouquet et al., 2007], B [Bouquet et al., 2004a], or Z [Ambert et al., 2002], have been used to automatically generate test cases and test drivers. On the basis of this research, model-based testing is beginning to penetrate into the industrial practice of software validation. A number of researchers have proposed test strategies based on state machines [Cheng and Krishnakumar, 1993]. These are typically based on coverage criteria such as all-transitions coverage (ensuring that every transition has been tested at least once), all-transition-pairs coverage, full predicate coverage and all-paths coverage [Utting et al., 2012]. However, there are significant scalability problems with most of these proposals (for example, all-paths coverage and all-transition-pairs coverage are impractical for industrial-size applications). New techniques based on symbolic execution can deal partly with this problem if test strategies can be defined and supported within a unified design process.

2.3.2/ SCENARIO-BASED TESTING

The key idea of scenario-based testing is to drive test generation from the behavior model on the basis of use case scenarios [Dadeau and Tissot, 2009, Lettrari and Klose, 2001]. A scenario is a specific sequence of actions and interactions between actors and the system under test. These test scenarios are refining business use cases, and can be formalized with UML diagrams such as sequence diagrams and activity diagrams. Each scenario is assigned a value to define risks. Scenarios with high total risk number will be scheduled to be tested with high priority. Automated test generation techniques generate test cases by converting scenarios into thin-threads and attaching data and the expected results to the call sequence.

2.3.3/ SEARCH-BASED TESTING

Search-based testing [McMinn, 2011] concerns the use of meta-heuristic search technique, such as Genetic Algorithms (GA for short). Search-based algorithms define optimization issues where the key stone is a fitness function. The simplest form of an optimization algorithm is random search. Though this is not specific to critical and complex systems, guided or feedback-directed forms of random testing have shown to be effective ways of automating component testing [Gotlieb and Petit, 2006]. However, random search is very poor at finding solutions when those solutions occupy a very small part of the overall search space. Thus, test data may be found faster if the search is given some guidance, i.e a problem-specific fitness function. This is particularly the case in GA, where the test data are seen as a chromosome. Then some algorithms, inspired by

the Darwinian theory (selection, crossover and mutations), are executed to explore the state space and to find a solution satisfying the fitness function. In the context of model-based testing, UML state machines are the most frequently used diagram for GA-based approaches [Shirole and Kumar, 2013].

2.3.4/ FUZZING APPROACHES

Fuzzing is extensively used in vulnerability testing [Sutton et al., 2007]. Its aim is to introduce malformed or mutated data to trigger flawed code in applications. Two main fuzzing techniques exist: mutation-based and generation-based. Mutation fuzzing consist of altering a data following specific heuristic, while generation-based fuzzing consist of generating test cases from the input specification. For instance, Duchene [Duchene, 2014] proposed to model the attacker's behavior. The model is then driven by a genetic algorithm that evolves the SUT input sequences.

2.3.5/ TEST GENERATION STRATEGIES

We presents here the work on test generation strategies from models. We introduce the most used testing techniques.

RANDOM TESTING

The random selection criterion consist in randomly selecting an execution sequence as a test. In a few words, it consists in browsing the specification randomly (by choosing randomly a stimuli). This generation strategy is very easy to process but it does not guarantee a structural coverage of the specification.

Although the relevance of the generated tests can not be asserted by the random criterion and that no model coverage is ensured, studies suggest that this type of strategy can be effective in the case of generating a small number of tests within a short time [Duran and Ntafos, 1984].

Some studies have suggested some improvements of this criterion since it is particularly suitable for models representing many behaviors (over 1000 states and more than 4000 transitions for instance). Indeed, for this type of model, applying coverage criteria (criterion generating a priori a small number of tests) can cause the generation of a huge amount of test cases. The approach is adapted using the application of probability on the selection of input data. These probabilities are calculated using specific criteria and thus enable a guided random generation [Denise et al., 2012]. Another way to take advantage of the diversity provided by the random generation while allowing relevant generation is to use the principles from genetic and evolutionary algorithms [Iqbal et al., 2011].

STRUCTURAL COVERAGE

One way to control the number of tests while maintaining a sufficient level of quality is to define coverage criteria over the model. The early work concerning coverage criteria are derived from structural testing techniques (white-box testing), which consist in defining

peace of the source code that need to be covered by the tests. In the context of black-box testing, these criteria is not applied to the code itself but to the model. The use of these criteria enables to extract test objectives The test generation then consists in covering these test objectives by generating stimuli sequences. In some cases, a test can cover multiple objectives.

As given in [Beizer, 1990], it exists three types of structural criteria:

- Transitions and states coverage is intended for modelling languages based on LTS (UML state machine, IOLTS...). For example, the criterion *All states* ensure the coverage of each modeled state [Offut et al., 1999].
- The control flow criterion relates to the coverage of decisions, loop or paths modeled in the form of conditional expression. For instance, the criterion *Decision Coverage* (DC) consists in covering each modeled decision, i.e. that for each modeled Boolean expression, at least one test must evaluates it to true, and at least one test must evaluates it to false.
- The data flow criterion is based on the definition and the use of the variables [Frankl and Weyuker, 1988]. For example, the *All definitions* criterion ensures to cover at least once each read access variables of the model.

REQUIREMENTS COVERAGE

The purpose is to link requirements with model elements that can satisfy them. The test generation then aims to cover all requirements through the coverage of model elements which they are associated. Today there are two approaches.

The first is to express this concept in the modelling languages. For example, in the UML paradigm, the SysML requirements diagram enables to express this connection. The second is to link the model and requirements in test generation tools. For example, the CertifyIt^{TM} tool includes a solution by providing OCL annotations dedicated to the requirements expression. It is then possible to ensure a specific coverage of OCL expressions satisfying specific requirements.

2.3.6/ IN-THE-LOOP TESTING

Testing plays a major role during the validation of safety-critical embedded systems in a "In-the-Loop" context. We thus present related work concerning testing in this context.

MODEL-IN-THE-LOOP TESTING

In most companies, MiL testing of controllers is currently limited to running the controller for a small number of input signals that are often selected based on the engineers' domain knowledge and experience.

They are numerous work that raise the confidence of test suites using several techniques for MIL testing. The work presented in [Iqbal et al., 2012a, Matinnejad et al., 2013],

proposes to combine search-based (typically evolutionary algorithm) and adaptive random testing. This combination enables to explore the state-space of signals provided by the plant to the controller under test. Later, these work are extended in [Matinnejad et al., 2014] to improve the search-based technique with surrogate models. These work enable to explore efficiently the state-space of a system. Our work is different as we assume that the system's parameters are known. Moreover, we propose an approach based on high-level models.

The work of [Amalfitano et al., 2014] focuses on automating the verdict assignment in a MIL process. It claims that engineers often used manual verdict assignment by visually comparing the test results with the oracle. They thus propose a new automated verdict assignment based on visual testing techniques that allow the automatic verification of the testing results. Therefore, this work focuses on the last step of the testing activity, i.e. the verdict assignment.

HARDWARE-IN-THE-LOOP TESTING

As for MIL testing, HIL testing often relies on empirical studies that lead to hand made test sequences. However, we have to point out that similar black-box testing approaches exist for real-time systems. For instance, Iqbal et al. [Iqbal et al., 2015, Arcuri et al., 2010] propose a modelling methodology based on UML and MARTE, in which the UML model is automatically translated into environment simulators implemented in Java. However, this modelling approach does not deal with continuous aspects, and differential and algebraic equations (DAEs) are hidden to the engineers.

Other work, such as in [Benigni and Monti, 2011] focuses on testing systems in a HIL context. However, such work do not consider test generation and high-level modelling.

2.3.7/ TESTING ASSESSMENT

We propose to perform model-based testing from SysML models of the plant in a "In-the-Loop" context. This approach uses structural coverage of state machines: transitions, states, and decision coverage. Indeed, the structural coverage criteria are used in automatic test generation tools, where the model is sufficient to produce test cases, i.e. the model contains enough information to specify the system under test and its environment, and to automatically derive the test objectives as well as the related test cases. Moreover, to the best of our knowledge, there is no reported approach in the literature that supports continuous and discrete SysML modelling for simulation and testing purposes.

2.4/ SUMMARY

In this chapter, we have presented languages and techniques that enable to specify and to validate critical and complex systems using simulation and testing. To adopt a model-centric approach, we have decided to use SysML as high-level modelling languages and Modelica for rapid prototyping and plant simulation.

In this way, our framework allows system engineers to stay as close as possible of the initial design specifications when achieving all the steps of the development life-cycle.

Moreover, it takes advantage of both approaches by ensuring a model centric process enabling validation, simulation and testing from the earliest stage of design. To achieve that, the architecture and the discrete behaviour of the system are described by a SysML model, which is annotated with OCL and Modelica code to specify its discrete and continuous features. This model is used to automatically generate real-time Modelica program for simulation, and black-box test cases for validation. The generated test cases can be simulated using the generated Modelica program to validate the design model as well as the physical system itself. Therefore, as illustrated in Fig. 2.1, the proposed framework contributes both to MIL process (model against simulated environment), and to HIL process (physical system against simulated environment).

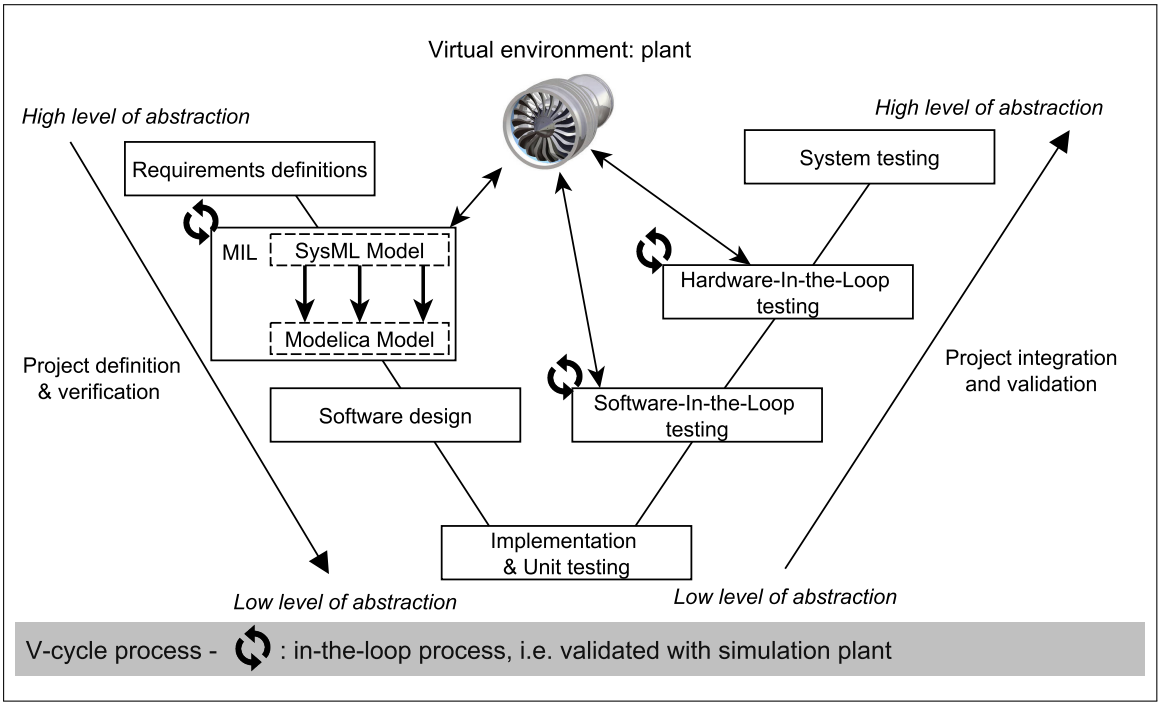


Figure 2.1: V-cycle Process Starting with SysML Model

II

CONTRIBUTIONS: DISCRETE AND CONTINUOUS MODELING IN SYSML

3

TECHNICAL BACKGROUND

Contents

This chapter introduces some technical background of the framework to achieve modelling, simulation, animation and test generation from SysML models. In this thesis, we propose to combine, in the same SysML model, continuous and discrete modelling for simulation and test generation respectively. For historical reason, we decided to use the CLPS-BZ solver to perform animation and test generation (see Sect. 1.2.1). Therefore, in the previous chapter we have focused on modelling and simulation language without considering solvers issue. We have concluded that SysML and Modelica are efficient languages for our needs of high-level modelling, simulation and test generation to validate critical and complex systems.

This dissertation describes an original SysML-based formal framework for simulation and testing of multi-physical and critical systems, that bridges the gap between high-level design model, starting point of MBSE approaches, and real-time execution platform, keystone of the In-the-Loop approaches. The proposed SysML based approach is an extension of the VETESS approach (see Fig. 1.2) and is illustrated in Fig 3.1. The first step of this approach (①) consists of modelling the SUT and the plant combining the SysML4MBT subset and the SysML4Modelica subset. The model is formalized by adding constraints expressed with a subset of the OCL for test generation purpose. In addition, a subset of the Modelica language is used as an action language to express continuous behaviours within state-machines.

The second step (②) consists of generating Modelica simulation code of the SUT and the plant model. At this stage, manual tests may be conducted to perform a first calibration of the model. Then the SysML model is transformed into a Constraint Satisfaction Problem (CSP) (③) to perform test cases generation over the plant model by applying structural and requirements coverage strategies.

After, the concretization step (④), test sequences are executed on the simulation model to perform MIL-testing. The model is calibrated to be as close as expected. Finally, the generated test sequences are executed on the real implementation. The simulation results (oracle) enable to perform back-to-back (⑤) testing during HIL-testing. Expected values are compared to obtained values in order to assign a verdict.

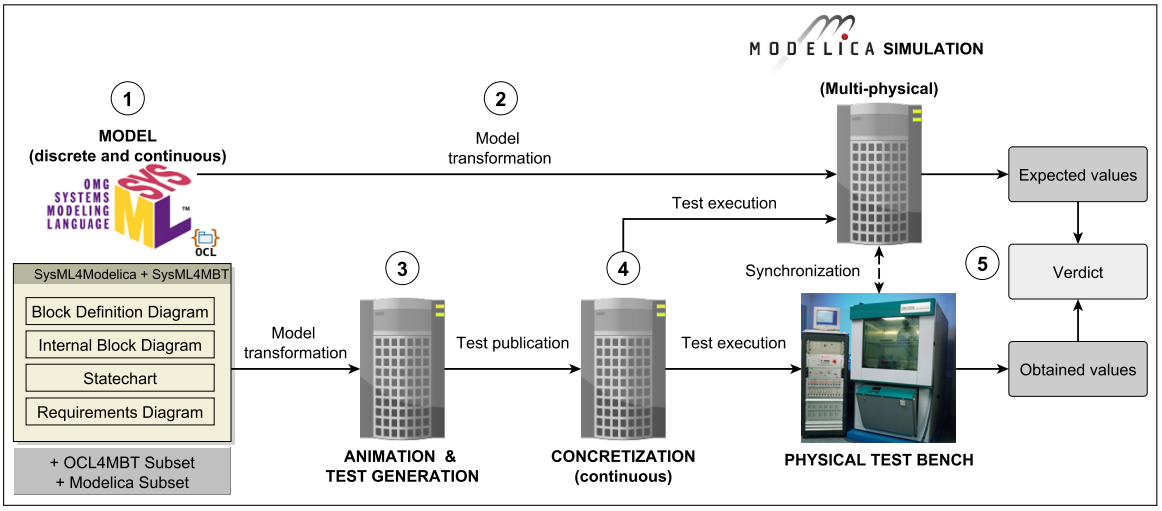


Figure 3.1: Overview of the Validation Process from SysML Models

In this chapter, we first present the SysML language in Sect. 3.1. Then, in Sect. 3.2, we present the Modelica language, the SysML4Modelica profile, and the OpenModelica environment. For animation and test generation purposes, we present the SysML4MBT subset with some backgrounds on the constraint solver CLPS-BZ in Sect. 3.3.

3.1/ THE SYSML LANGUAGE

SysML was first proposed by the INCOSE (International Council on Systems Engineering) at the beginning of the 21st century. The aim was to address issues of the system engineering field by proposing a UML based language. Since 2006, SysML is normalized by the OMG. The version 1.0 was defined in 2007 and the SysML 1.4 specification was adopted in March 2014 but in this thesis, we have used the SysML 1.3 specification. As depicted in Fig 3.2, SysML adapts the UML semantic by modifying (SysML profile) or reusing some UML elements (UML4SysML).

Just as UML, SysML allows the specification, analysis, design, verification and validation of many systems. However, SysML is dedicated to system engineering business processes that use to model automotive or aeronautic systems including hardware and software components. In concrete terms, this standard takes 7 of the 13 UML 2.0 diagrams (among these, three are modified) and includes two new diagram types. These diagrams can be grouped into three categories. First, the requirements diagram is used to represent the system requirements. Then, there are four structural diagram types. They allow to give the static view of a system. Finally, four behavioral diagrams are used to represent the dynamic view of a system. Figure 3.3 depicts the position of each SysML diagram relating to UML2. Some SysML diagrams are the same as UML. They are represented by the boxes in thin lines. Then some diagrams exist in both languages but have been modified for SysML. They are represented by the boxes in dotted lines. And finally, the new SysML diagrams are represented by the boxes in thick lines.

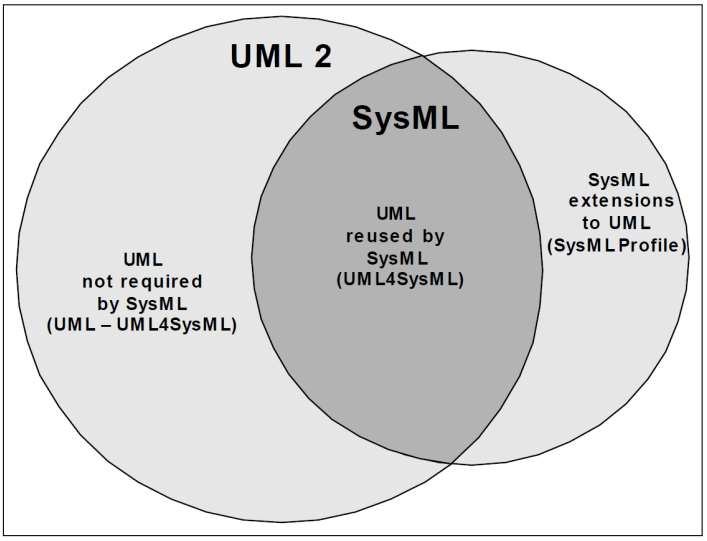


Figure 3.2: Overview of SysML/UML Interrelationship

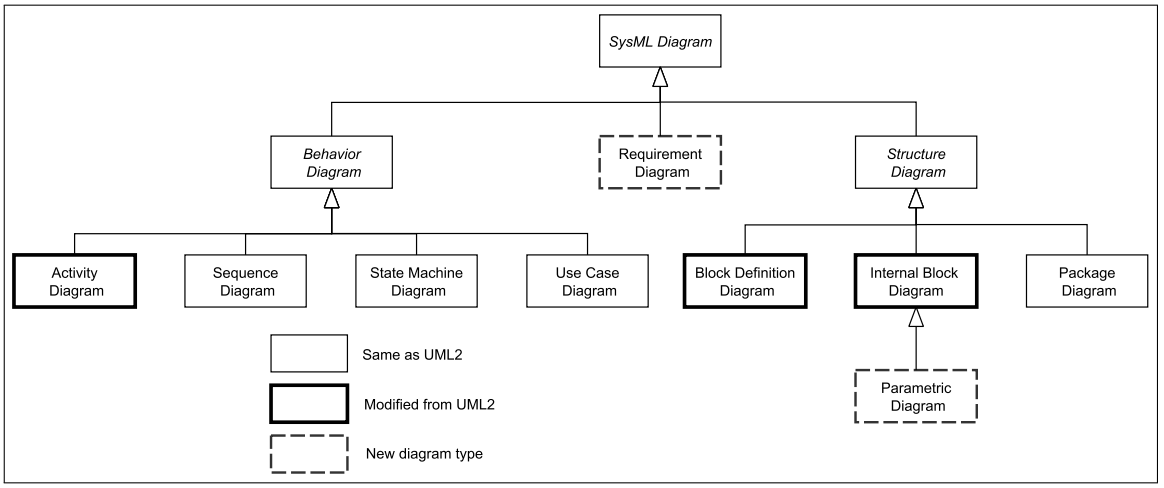


Figure 3.3: Overview of the SysML Diagrams

We now present an overview of each SysML diagram. First, we present the structural diagram in the part 3.1.1. Then, we introduce the behavioral diagrams in the part 3.1.2. Finally, an overview of the requirements diagram is given in the part 3.1.3.

3.1.1/ STRUCTURAL MODELLING

The structural diagrams allow the representation of the static view of a system. SysML proposes four diagrams: the BDD (Block Definition Diagram), the IBD (Internal Block Diagram), the parametric diagram, and the package diagram. While the BDD and the IBD are based on existing UML diagrams, parametric diagram is unique. The package diagram is identical to UML2. This section briefly presents them in that order.

BLOCK DEFINITION DIAGRAM

The Block Definition Diagram (BDD) enables to graphically represent the composition, aggregation and generalization between blocks, attributes and operations. This diagram is equivalent to the UML class diagram. Structural modelling is mainly done with the *Block* element. It inherits from the UML class concept but it has been adapted to add physical system semantic.

The UML class concept is enriched by the concept of SysML block to adapt the semantic to the field of Systems Engineering. Thus, since a UML class represents a software part, a SysML block includes software concepts, hardware, data and also process.

INTERNAL BLOCK DIAGRAM

The Internal Block Diagram (IBD) represents the interconnections between instances of blocks, named parts, through *FlowPorts*. *FlowPorts* are introduced by SysML as a new type of ports. These elements enable to specify physical interfaces where a continuous exchange of information and physical energy take place. The IBD is based on the UML components diagram. We note that the IBD is very close to the coupled model formalism [Vangheluwe et al., 2002] defined as follow: $CM =< \eta, \Phi, S, C >$, where η is the unique identifier of the model, Φ is the set of ports to the outside, S is the set of sub-models that compose CM and C is the coupling information contained in a graph structure. The coupled model CM is said non-causal and continuous if the graph C is undirected and causal if C is directed (i.e. ports are directed with $\{$*in, ou, inout*$\}$ causalities).

PARAMETRIC DIAGRAM

Equation modelling is possible within SysML by the use of the parametric diagram. It is a new type of diagram over UML that enables to specify mathematical expressions between model elements using constraint blocks. Constraint blocks enable to specify mathematical expressions in which each parameter can refer to an element of the model (block property for instance).

PACKAGE DIAGRAM

The package diagram is the same as UML. It shows the general organization of the model enabling a simple representation of the packages used in the model and the various links between them.

3.1.2/ BEHAVIORAL MODELLING

Behavioral diagrams enable to represent the dynamic view of a system. In SysML, there are four behavioral diagrams: the use case diagram, the sequence diagram, the activity diagram, and the state machine diagram. Apart from the activity diagram that has been extended (compared to the UML version) these diagrams come from the UML notation.

USE CASE DIAGRAM

The use case diagrams are used to represent the functionalities and operations that the system should provide to users. It enables to specify the actors (human or machine) and their interactions with the system through use cases. The use case diagram is often established by the project owner during the specifications drafting.

SEQUENCE DIAGRAM

The sequence diagram is used to represent a sequence of actions that are represented in the use case diagram. The sequence diagram models the sequence of interactions between system elements or between the system and its environment. To model a scenario, each actor is represented using lifeline. Each lifeline contains actions that are interconnected to describe the sequence of the scenario.

ACTIVITY DIAGRAM

The activity diagram models the flow of information and the flow of activities of the system. It enables to graphically describe a process (or the flow of a use case) in terms of sequences of activities. It permits to design or to document the behavior or any designed element. Compared to the UML activity diagrams, SysML added the following aspects: modelling of exchange and continuous behavior, it supports the control of an activity during its execution, and it redefines activities as belonging to a block.

STATE MACHINE DIAGRAM

State machines are also included in the SysML language without the UML concept of protocol state machines. The state machine diagram is used to represent the behavior of a block. It models the possible states of the component. The state machine behavior is driven by stimuli. A transition can be automatic when no triggering events are specified. In addition, Boolean expressions written in natural language or using Object Constraint Language (OCL) enable to constrain transitions. The notation based on these structures are very popular in the research field and in the industrial world [Cheng and Krishnakumar, 1993].

3.1.3/ REQUIREMENTS MODELLING

SysML proposes to model requirements through a new diagram. Requirements blocks enable to integrate textual requirements connected with hierarchical relationships and traceability links. The requirements are referred to as SysML cross members because they are connected to different elements of the language. Thus, it is possible to know the elements that model and satisfy the requirements. This requires the modelling and the verification and validation activities at the soonest in the life cycle of the system.

A SysML requirements is a stereotyped class. Several types of relations between requirements exist:

- Trace relation: every other relations generalize *Trace*.
- Derive relation: binds a derived requirements to its source requirements. A derived requirements generally corresponds to the requirements of the next hierarchical level of the system.
- Satisfy relation: connect a model element to a requirements ensuring traceability of requirements.
- Copy relation: specifies that a requirements is a copy of an existing one, allowing the reuse of the requirements in another context.
- Verify relation: defines how a test case verifies a requirements.
- Refine relation: describes how a set of SysML elements may be used to refine a requirements.

3.1.4/ THE SYSML4MBT SUBSET

SysML4MBT is a SysML subset dedicated to model-based testing activity for physical and embedded systems validation. It was proposed during the VETESS project (see Sect. 1.2.2) and applied to test and validate a steering column, wipers, and an electronic car seat. SysML4MBT considers the following SysML diagrams:

- the block definition diagram with blocks, attributes, operations, and associations,
- the internal block diagram with parts, flow ports, and connectors,
- the state machines,
- a subset of OCL4MBT.

SysML4MBT enables to perform test generation using CertifyIt^{TM} by applying structural coverage criteria (coverage of states, transitions, etc), and a new criteria dubbed com-cover. The com-cover criteria is specified thanks to a new OCL operator for signal exchange.

This section has introduced the SysML language and the SysML4MBT subset. The next section presents the Modelica language. This simulation language was selected to be the target of model transformation and code generation from SysML models.

3.2/ THE MODELICA LANGUAGE

Modelica is a simulation language that has been developed since 1996 by the (non-profit) Modelica Association. Modelica is used in industry since year 2000. In this section, we present the object-oriented modelling approach proposed by Modelica. Then, we introduce the concept of equation modelling. Finally, after presenting the OMG's SysML-Modelica Transformation specification, we give some details about the OpenModelica simulation environment.

3.2.1/ OBJECT-ORIENTED MODELLING

Modelica is an object-oriented modelling language for simulation purpose. A system is represented by a diagram, which consists of connected components, such as a resistor, or a hydraulic cylinder. A component has *connectors*, (also called *ports*) that enable to describe the possible interactions, e.g., an electrical pin, a mechanical flange, or an input signal. A diagram model can be constructed by drawing connection lines between connectors. An example of such diagram is given in Fig 3.4.

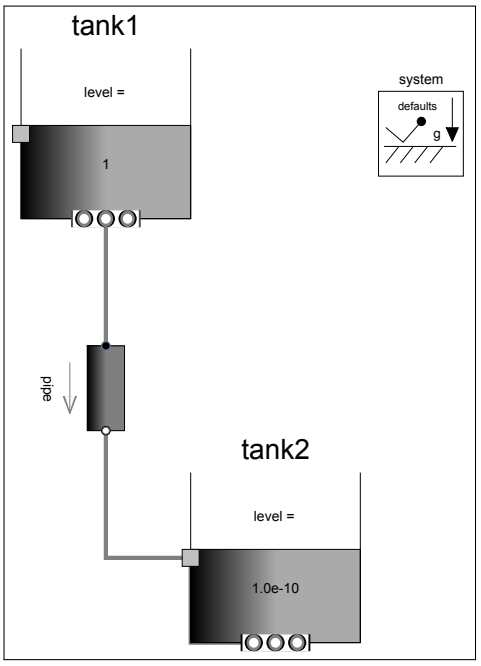


Figure 3.4: Modelica Diagram Example of a Tanks System (from OpenModelica Library)

The fundamental structuring unit of modelling is the class. Classes provide the structure for objects that can contain parameters, variables, constants, equations, and algorithms. Essentially everything in Modelica is a class, from the predefined classes *Integer* and *Real*, to large packages such as the Modelica standard library. Hence, Modelica natively supports inheritance.

Modelica also defines specialized kinds of classes: *record*, *type*, *model*, *block*, *package*, *function*, and *connector*. These specialized classes have the properties of a general class. The Table 3.1 summarizes the definition of the specialized classes.

3.2.2/ EQUATION MODELLING

The main goal of Modelica is to model the dynamic behaviour of systems composed of parts from mechanical, electrical, thermal, hydraulic, pneumatic, fluid, control and other domains. Models in Modelica are mathematically described by differential, algebraic and discrete equations (hybrid DAEs). Modelica does not offer a way to describe partial differential equation, i.e., it does not support FEM (Finite Element Method). Modelica is designed such that specialized algorithms can support models that have more than one hundred thousand equations. In this way, Modelica is suited and used for *In-the-Loop* simulation [Ferreira et al., 1999] and for embedded control systems.

Table 3.1: Specialized Classes

Class	Definition
record	Only public sections are allowed in the definition or in any of its components (i.e., equation, algorithm, initial equation, initial algorithm and protected sections are not allowed). Record components can be used as component references in expressions and in the left hand side of assignments, subject to normal type compatibility rules. May only contain components of specialized class record and type.
type	This kind of class may only be predefined types, enumerations, array of type, or classes extending from type.
model	Identical to the basic class concept.
block	Identical as *model* with the restriction that each connector component of a block must have prefixes *input* or *output* for all connector variables.
package	May only contain declarations of classes and constants.
function	May only contain parameters that have *input* prefix for input parameter and *output* prefix for result parameter. A function can't be used in connections.
connector	Only public sections are allowed in the definition or in any of its components (i.e., equation, algorithm, initial equation, initial algorithm and protected sections are not allowed). This specialized class serves to specify what is flowing between two components: it types components ports.

Let us consider the simple differential equation (1) of the running example (see Sect. 1.5):

$$\dot{h} = \frac{f_{\text{in}} - f_{\text{out}}}{s} \tag{1}$$

This equation can be represented in Modelica as follows:

```
der(h) = (f_in.value- f_out.value) / s;
```

Since the variable h in the equation is a continuous real valued variable, its declaration in Modelica takes the form `Real h;`. The `Real` type is one of the Modelica primitive types. Once all variables have been declared, we can write the equations that describe the behaviour of the model. In this case, we can use the `der` operator to represent the time derivative of h. To simulate a model in Modelica, the number of variables must equal the number of equations and the number of equations must be fixed during the simulation.

Unlike most programming languages, Modelica code can't be interpreted as a set of instructions to be executed one after the other. Instead, a Modelica compiler transforms the model into something that we can simulate. This simulation step essentially amounts to solving (usually numerically) the equation and providing a solution trajectory.

As we have seen, Modelica allows us to describe model behaviour in terms of differential equations. But the initial conditions we choose are as important as the equations. For this reason, Modelica also provides constructs for describing the initialization of our system of equations. For example, if we wanted the initial value of h in our model to be 2, we could add an initial equation section to our model as follows:

```
initial equation
  h = 2;
equation
 der(h) = (f_in.value- f_out.value) / s;
```

In the previous example, the initial value of h at the start of the simulation was unspecified. Generally speaking, this means that the initial value for h will be the value of its *start* attribute (which is zero by default). However, because each tool uses their own specific algorithms to formulate the final system of equations, it is always best to state initial conditions explicitly, as we have done here. By adding this equation to the initial equation section, we are explicitly specifying the initial condition for h.

The last important feature in Modelica concerns variables and variability. A model definition typically contains variable declarations. Within Modelica, it is possible to declare four kinds of variables: *parameter*, *constant*, *discrete*, and *continuous*. By default, variables declared inside a model are assumed to be continuous variables (but which may also include discontinuities). However, it is also possible to add the parameter qualifier in front of a variable declaration and to indicate that the variable is known a priori.

Closely related to the parameter qualifier is the constant qualifier. When placed in front of a variable declaration, the constant qualifier also implies that the value of the variable is known a priori and is constant with respect to time. The distinction between the two lies in the fact that a parameter value can be changed from one simulation to the next whereas the value of a constant cannot be changed once the model is compiled. A constant is frequently used to represent physical quantities like π or the Earth's gravitational acceleration, which can be assumed constant for most engineering simulations.

Unlike continuous variables, discrete variables change their value at specific time and have time derivative to zero. Such variables are declared using the discrete qualifier. For more precision about hybrid DAEs in Modelica we kindly refer the reader to the paper of Lundvall et al. [Lundvall and Fritzson, 2005].

3.2.3/ THE SYSML4MODELICA PROFILE

The OMG is working on the SysML-Modelica transformation [OMGSM, 2012] since 2008 (version 1.0 from November 2012 accepted as current specification). The objectives of the SysML-Modelica Transformation specification are to enable and specify bi-directional transformation between the two modelling languages. This specification integrates semantic connections between SysML and the Modelica simulation language, using UML profiling technique. The specification gives an extension to SysML, called SysML4Modelica, which proposes matching semantics between SysML4Modelica constructs and the Modelica language. The integration of Modelica concepts into SysML is based on a meta modelling approach as depicted on the right-hand side of Fig. 3.5.

Basically, the SysML4Modelica constructs enable to stereotype elements that are part of the Block Definition Diagram (BDD) and the Internal Block Diagram (IBD) of SysML. Hence, the SysML4Modelica profile enables to bridge the gap between two modelling language: SysML, which is a non executable graphical high level modelling language, and Modelica, which is used as simulation language for complex and heterogeneous systems. Translating SysML-based specifications into Modelica environment enables rigorous static and dynamic system analysis.

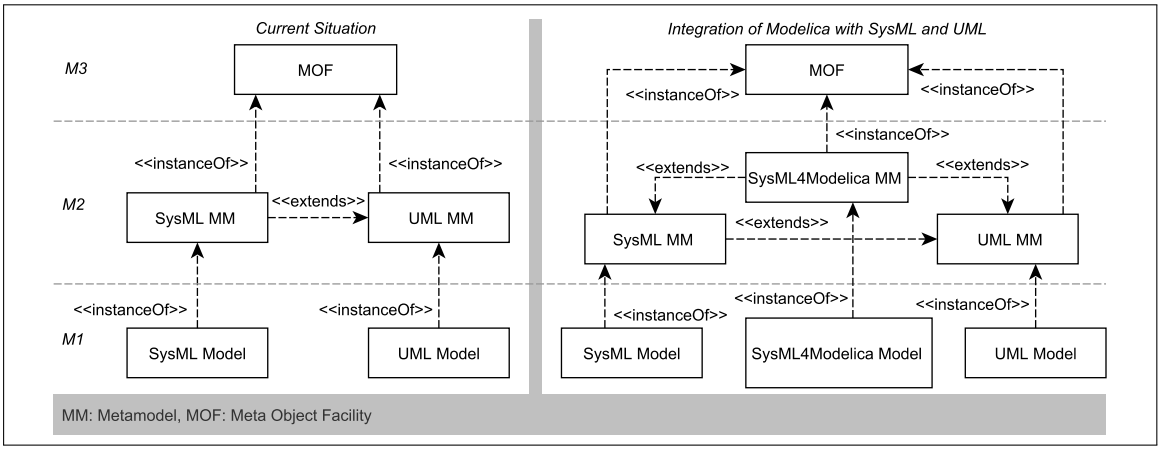


Figure 3.5: SysML4Modelica Profile over UML and SysML

3.2.4/ THE OPENMODELICA ENVIRONMENT

OpenModelica [1] is an open-source Modelica-based modelling and simulation environment for industrial and academic usage. It has been developed by the OSMC (Open Source Modelica Consortium) since year 2007. OSMC is a non-profit, non-governmental organization with the aim of developing and promoting the development and usage of the OpenModelica open source implementation of the Modelica modelling language and OpenModelica associated open-source tools and libraries, collectively named the OpenModelica Environment.

The current version of the OpenModelica environment allows most of the expression, algorithm, and function parts of Modelica to be executed interactively, as well as equation models and Modelica functions to be compiled into efficient C code. The generated C code is combined with a library of utility functions, a run-time library, and a numerical DAE solver.

The modeled system is supposed to be operating in continuous time, i.e, the input, state and output variables change continuously. Nevertheless, the simulation program has the values only at t_i and it must estimate the values at t_{i+1} without knowledge of what happening between t_i and t_{i+1}. Solving this issue is generally known as numerical integration methods [Conte and Boor, 1980]. Modelica feats several integration methods such as *euler*, *rungekutta*, *dassl* among others. Within SysML, we do not have to deal with integration issues (numerical solvers do the job) but we have to preserve the continuous paradigm to enable simulation.

Figure 3.6 depicts the overall OpenModelica environment. Arrows denote data and control flow. The interactive session handler receives commands and shows results from evaluating commands and expressions that are translated and executed. Several subsystems provide different forms of browsing and textual editing of Modelica code. The debugger currently provides debugging of an extended algorithmic subset of Modelica.

[1] https://www.openmodelica.org/ [Last visited in June 2015]

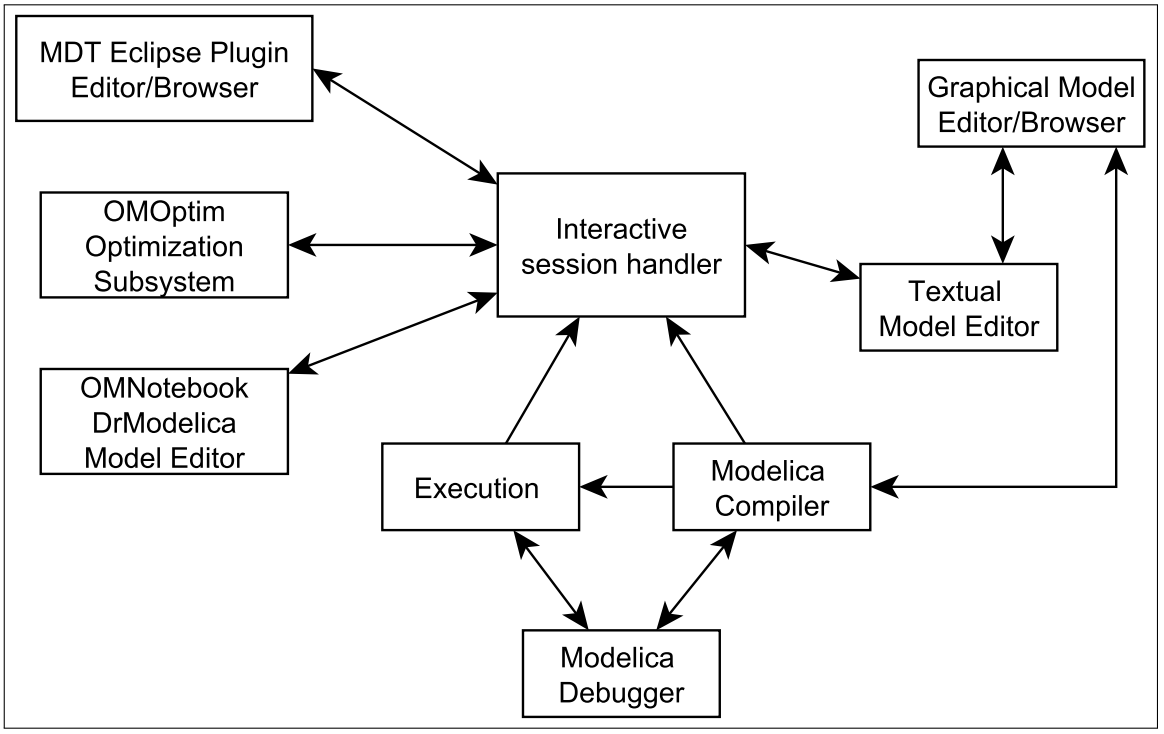


Figure 3.6: Architecture of the OpenModelica Environment

In this thesis, we used the OMEdit graphical and textual model editor to perform simulation of the generated Modelica models. OMEdit is a graphical connection editor, for component based model design by connecting instances of Modelica classes, and browsing Modelica model libraries for reading and picking component models. The graphical model editor also includes a textual editor for editing model class definitions, and a window for interactive Modelica command evaluation.

3.3/ CLPS-BZ AND BZP

In this section we present the solver CLPS-BZ and the BZP file format. We decided to use this solver because it is developed in the DISC department since 1990 [Legeard and Legros, 1991]. CLPS has been extended at the end of the 90's to support B and Z specifications.

3.3.1/ THE CLPS-BZ SOLVER

As depicted in Fig. 3.7, CLPS-BZ [Bouquet et al., 2004a] (Constraint Logic Programming with Set for B and Z) is a constraint solver that augments the capabilities of (and co-operates with) the CLP(FD) library (integer finite domain solver of SICStus Prolog[2]) by handling constraints over sets. Then, the work published in [Legeard and Py, 1999, Legeard et al., 2002] and [Legeard et al., 2004] permitted to add a translation layer to support support B and Z operators.

[2]https://sicstus.sics.se

Later, it has been extended to manage object-oriented specifications, such as UML with OCL constraints. Basically, such models are translated into an internal Prolog-readable syntax, called BZP, which provides special constructs for defining UML diagrams and OCL expressions as constraints over sets.

CLPS-BZ makes it possible to efficiently execute on discrete domains the BZP code, both for model animation and for test computation. Test computation consists to look into the graph of reachable states of the system described by the constraints to achieve classical test coverage criteria including transition-based, decision based and data-oriented criteria [Ambert et al., 2013]. Afterwards, a set of execution traces, that define the test cases, are computed by solving the constraints to find the sequences of operation invocations that ensure the given criteria. To achieve that, CLPS-BZ animates the model and computes a reachability graph, whose nodes are the constrained states built during the animation, and whose transitions define an operation invocation. Using constraint solving dramatically reduces the search space during test generation, which allows the method to scale to larger systems.

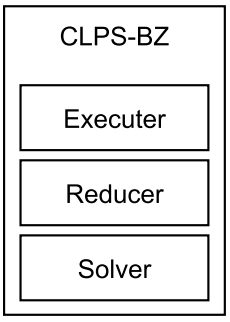


Figure 3.7: Overview of the CLPS-BZ Solver

3.3.2/ THE BZP FORMAT

The solver, named CLPS-BZ, uses BZP file as input specification [Bouquet et al., 2004c]. The BZP format is a Prolog-readable syntax which provides special constructs for defining state machines and operations. A BZP specification contains an unordered set of facts, each of which has one of the following forms:

- *model(*η*)*, where η is the model's name.
- *context(*ν*)*, where ν is the context's name. This fact enables to define an object-oriented structure. For instance, each class and each state define a new context.
- *declaration(*ι*,* ν*,* δ*,* $x(\epsilon)$*,* ω*,* Σ*)*, where:
  - ι is a unique identifier,
  - ν is the name of the context to which the declared data relates,
  - δ is the kind of the declared data $(static, variable, input, output, local)$,
  - $x(\epsilon)$: ϵ is the name of the declared data. Additionally, x takes the following value: s, v, i, o depending on δ.
  - ω is the type of the declared data and is defined with the following grammar: $Type ::= atom \mid int \mid set(Type) \mid pair(Type, Type)$.
  - Σ is the set of values in which the data can take a value.
- *predicat(*ν*,* π*,* ι*,* ρ*)*, where:
  - ν is the name of the context to which the predicate is related,
  - π is the type of the predicate $(static, invariant, initialisation, pre, post)$,
  - ι is a unique identifier,
  - ρ is the expression of the predicat.

- *operation(ι, ν, o)*, where:
  - ι is a unique identifier,
  - ν is the name of the context to which the operation relates,
  - o is the operation's name.
- *event(ι, ν, υ)*, where:
  - ι is a unique identifier,
  - ν is the name of the context to which the event relates,
  - υ is the event's name.

3.4/ SYNTHESIS

In this chapter, we have presented the SysML language, the Modelica simulation language, and the CLPS-BZ solver. The SysML language is a high-level modelling language that is the starting point of our approach. Therefore, we propose a framework that aims (1) to avoid managing several models (at least one for high-level discrete design and one for low-level continuous features) that require to be manually synchronized, (2) to increase the automation level of the model-based testing approach by minimizing the number of testing artefacts and by providing a native link between abstract data (from SysML structures) and executable structures (derived from Modelica code), and (3) to foster the use of MBSE approach by supporting in the same modelling framework all design steps of the real-time system life-cycle activities.

More precisely, the SysML model is designed from requirements and combines discrete and continuous aspects. The model is then the entry point of a translation to the BZP format for the CLPS-BZ solver. However, the CLPS-BZ technology is only able to handle constraints on discrete domains and thus can execute neither animation nor test generation based on continuous formula to efficiently address real-time systems. Therefore, CLPS-BZ enables to derive test cases, as sequences of operation invocations, but an other and independent model or program is necessary to execute them in the continuous domain to gather the real and expected results. Moreover, SysML is natively not executable: it does not include an action language, which could allow to simulate SysML model, and even less if equations occur.

To overcome this lack, the OMG has proposed an extension to SysML to allow clarifying such mathematical properties into SysML models using Modelica code. Hence, we propose to use this extension, called SysML4modelica, to adapt and complete the existing approach to be able to manage in a single model both high-level discrete requirements and low-level continuous behaviours for test generation purpose.

The next two chapters precisely describes the modelling framework we defined to efficiently combine discrete and continuous features in a single model for simulation and model-based testing purposes. In the rest of this thesis, to dispel any ambiguity and avoid misunderstanding, *animation* is defined as a discrete evaluation of the model (variables belong to finite domains or sets), i.e. an execution of the model based on constraint solving restricted to the SysML data that belong to finite domains or sets, whereas *simulation* means real-time simulation of the model, i.e. an execution of the Modelica code describing the system in a continuous-time process.

4

SysML MODELLING FOR SIMULATION

Contents

This chapter gives a formal description of the SysML modelling framework from the continuous point of view. We first describe in Sect. 4.1 an overview of the OMG's SysML4Modelica profile subset that is most used in this thesis. We also propose to extend this subset to take into account elements for test generation. Then, we present in Sect. 4.2 the formal modelling framework that enables to model continuous part of a system for continuous simulation within the OpenModelica simulation environment. Finally, we illustrate in Sect. 4.3 this framework with the running example given in Sect. 1.5.

4.1/ SysML for Modelica Simulation

This section introduces the SysML4Modelica elements for continuous simulation. We focus on the SysML4Modelica elements given by the SysML-Modelica Transformation specification [OMGSM, 2012], i.e elements that are part of the BDD and the IBD. Then, we present the motivations for the extension of the SysML4Modelica subset. This extension includes SysML state machines and sequence diagrams.

4.1.1/ SysML4Modelica Profile

The SysML4Modelica profile proposed by the OMG enables to give Modelica semantic to BDD's elements and IBD's elements. In addition, it specifies stereotypes for SysML FunctionBehaviour and their associated parameters.

Block Definition Diagram Profiling

The main SysML BDD element is the block. Within SysML4Modelica, a SysML block may take one of the 6 following stereotypes: `<<modelicaClass>>`, `<<modelicaModel>>`, `<<modelicaBlock>>`, `<<modelicaRecord>>`, `<<modelicaConnector>>`, and

<<modelicaPackage>>. Indeed, each of these stereotypes is a generalization of the <<modelicaClassDefinition>> concept. This is due to the fact that, the fundamental structuring unit in Modelica is the class (see Sect. 3.2.1). Therefore, each of these stereotypes give a different semantic (as described in the Table 3.1), which implies different modelling rules.

We have to make the distinction between <<modelicaConnector>> and the other above stereotypes. Indeed, <<modelicaConnector>> also stereotypes blocks but these blocks are not components. <<modelicaConnector>> stereotyped blocks are used for flow ports typing. They serve to describe what flows between two components (energy, flow of matter, etc).

A SysML block can contain attributes. Within SysML4Modelica, an attribute becomes a <<modelicaValueProperty>>. This stereotype makes sense since we need need to know at least the variability of each property owned by a component (see Sect. 3.2.2): parameter, constant, discrete, and continuous.

The SysML4Modelica profile proposes also to extends the UML generalization with the <<modelicaExtends>> stereotype. The only difference is that in Modelica, the class being extended can be locally modified.

Finally, the SysML constraint element is extended with the <<modelicaEquation>> stereotype. This stereotype contains a Boolean attribute *isInitial*. This attribute is true when the equation represents an initial equation section in Modelica.

INTERNAL BLOCK DIAGRAM PROFILING

In the Modelica language, instances of a class are referred to as *Components*. In SysML, these can be mapped to Block Properties, such as Value Property, Part Property, or Flow Port. Modelica does not distinguish explicitly between Value Properties, Parts, or Ports. Instead, whether a component is interpreted as a Value Property, Part or Port depends on the restricted type to which the component has been typed. If the component is of restricted type class, model, or block then it is mapped to a <<modelicaPart>>; if it is of restricted type connector then it is mapped to a <<modelicaPort>>; and if it is of restricted type record or type then it is mapped to <<modelicaValueProperty>>.

Connecting two Parts through Ports is possible using Connector element. SysML4Modelica offers to specify such connection using the <<modelicaConnection>> stereotype. This stereotype is applied to Connector elements.

SYSML FUNCTIONBEHAVIOR AND PARAMETER PROFILING

Within Modelica, it is possible to represent a callable section of procedural algorithmic code without side effects using the specialized class Function. Therefore, SysML4Modelica offers the possibility to declare such functions using the <<modelicaFunction>> stereotype applied to SysML FunctionBehavior. In addition, a Modelica restricted class function, can contain Modelica parameter declarations. Then, an equivalent stereotype for function's parameters is created: <<modelicaFunctionParameter>> .

4.1.2/ EXTENSION PROPOSAL

In this thesis, we have extended the mapping to take into account enumerations, state machines, and sequence diagram. Indeed, state machines are useful to specify specific states of a (sub)system, whereas sequence diagrams enable to specify sequence of actions over the system. This extended mapping does not rely on a profiling approach.

ENUMERATIONS

Enumeration types exist both in SysML and in Modelica. SysML enumerations enable to declare abstract type for block attributes. Within Modelica, a declaration of the form

```
type E = enumeration([enum_list]);
```

defines an enumeration type E with its associated enumeration literals of the enum-list. Just like SysML, the enumeration literals shall be distinct within the enumeration type.

STATE MACHINES

Concerning state machines, we have considered regions, states, transitions, triggers, guards, and effects (we have excluded join, fork and history pseudo-states). We propose to translate state machines to sequential Modelica algorithms containing exclusively `when` statements. Such algorithms enable to know in which state the (sub)system is at a specific time. A state is declared as a Modelica Boolean and a transition as a Modelica `when` statement. A `when` statement is activated whether its conditional expression is true. In this context a conditional expression of a `when` statement is as the following grammar:

$whenCS$: $transition.source$ [**and** $guard$] [**and** $trigger$].

Therefore, transition's guard and trigger define a Boolean expression of a `when` statement. The following piece of Modelica code depicts an example of a transition between two states $S1$ and $S2$. This transition contains a trigger, a guard (written with the Modelica syntax), and an effect (written with the Modelica syntax). Note that $onEntry$ and $onExit$ behaviours are also written with the Modelica syntax within the SysML state machine.

```
when S1 and trigger and guard then
  S1 := false;
  S2 := true;
  S1.onExit statement
  effect
  S2.onEntry statement
end when;
```

SEQUENCE DIAGRAM

We have defined mapping rules between sequence diagram and Modelica. These rules enable simulations of test cases specified with UML sequence diagrams. Interaction is transformed into a Modelica model containing Modelica components derived from life lines. A test case is then specified with a Modelica algorithm that is derived from asynchronous messages and duration constraints on action execution specification. It's typically `when` statements on time events that trigger specific actions of the system. Combined fragment are not considered yet. It would be part of future work to specify more complex test cases.

4.1.3/ SUMMARY

Table 4.1 summarizes the mapping between SysML elements and Modelica constructs using the SysML4Modelica profile. It contains elements that are part of the BDD and IBD. Then, Table 4.2 summarizes the extended mapping between state machines, sequence diagram and Modelica constructs. These concepts are illustrated in Sect. 4.3 with the running example. The next section introduces the SysML formal modelling framework for continuous simulation.

Table 4.1: Mapping for the SysML4Modelica Stereotypes

SysML Base Class	SysML4Modelica	Modelica Construct
Block Definition Diagram		
UML4SysML::Classifier	`<<modelicaClassDefinition>>`	Abstract generalization for Modelica Class
SysML::Blocks::Block	`<<modelicaModel>>`	Class and Model
SysML::Blocks::Block	`<<modelicaRecord>>`	Record
SysML::Blocks::Block	`<<modelicaBlock>>`	Block
SysML::Blocks::Block	`<<modelicaConnector>>`	Connector
SysML::Blocks::Block	`<<modelicaPackage>>`	Package
UML4SysML::Property	`<<modelicaValueProperty>>`	Variable
UML4SysML::Generalization	`<<modelicaExtends>>`	Extends statement
UML4SysML::Constraints	`<<modelicaEquation>>`	Equation
UML4SysML::FunctionBehavior	`<<modelicaFunction>>`	Function
UML4SysML::Parameter	`<<modelicaFunctionParameter>>`	Parameter
Internal Block Diagram		
UML4SysML::Property	`<<modelicaPart>>`	Class instance
SysML::Ports&Flows::FlowPort	`<<modelicaPort>>`	Port
UML4SysML::Connector	`<<modelicaConnection>>`	Connect statement

Table 4.2: Extended Mapping for Modelica Simulation

SysML Base Class	Modelica Construct
UML4SysML::Enumeration	Enumeration type
State machine	
UML4SysML::Region	algorithm section
UML4SysML::PseudoState (Initial state)	**when initial** statement
UML4SysML::State	Boolean variable
UML4SysML::State onEntry	Modelica statement
UML4SysML::State onExit	Modelica statement
UML4SysML::Transition	**when** statement
UML4SysML::Transition guard	Boolean expression
UML4SysML::Transition effect	Modelica statement
UML4SysML::Transition event trigger	Boolean variable
Sequence diagram	
UML4SysML::Interaction	Modelica Model
UML4SysML::Lifeline	Modelica component
UML4SysML::Message	**when** statement
UML4SysML::Interaction duration constraint	time event

4.2/ FORMAL MODELLING FRAMEWORK

This section presents a formalization of the SysML subset for continuous modelling. This part includes a first section that presents the structures that we will use in the formalization and several sections containing the formalization itself.

4.2.1/ FORMAL STRUCTURES

This section presents the structures used in the definition of the SysML formal framework for simulation. Different structures such as tuples, sets or sequences are used to formalize the various modelling elements.

TUPLES

A tuple is generally an orderly structure of fixed size, which can contain multiple elements. The various elements are not necessarily of the same type. The type is defined in the construction.

For example, a 2-tuple named $Tuple$ will be written $Tuple =< P1, P2 >$. To know the value of $P2$, we will use the statement $Tuple.P2$.

Finally, we can instantiate each of these structures. For instance, let $T1_{Int}$ be a 2-tuple containing integers, then $T1_{Int}$ is defined by $T1_{Int} =< 4, 12 >$. Moreover, $T1_{Int}.P1 = 4$ and $T1_{Int}.P2 = 12$.

SETS

A set refers to a collection of objects. The traditional operators, such as union ($\cup$), intersection ($\cap$), the symbol belongs ($\in$), and so on, can be used. A set is not ordered and can not contain duplicates.

To define a set, the syntax $\Gamma = \{a, b, c \ldots\}$ is used in this thesis. For instance, the notation $\Gamma_{Int} = \{1, 2, 3\}$ defines a set of three integers. The empty set is represented by $\emptyset$.

4.2.2/ MODEL FOR SIMULATION

The SysML subset for simulation purpose focuses on the following diagrams: BDD, IBD, state machine and sequence diagram. The structural view of the system is specified in the BDD with blocks, which are connected each other using flow ports that are depicted in the IBD. The behaviour of each system component may be described using state machines and constraints. Sequence diagram may be used to activate state machines over time.

We define a SysML model for simulation as a model M_s comprising two kind of blocks (blocks for component definition and blocks for flow ports typing), enumerations, and interactions. A block that types flow ports only contains properties (no behaviour). SysML enumerations enable declaring abstract types that can be used during a Modelica simulation. Interactions are used to specify sequence of actions.

Definition 1: Model for simulation

Let M_s, the model for simulation, be given by $M_s =< \eta, \Gamma_{Bs}, \Gamma_{Bf}, \Gamma_{Enum}, \Gamma_{Inter}, \tau_a, \tau_s >$, where:

1. η is the name of the model,
2. Γ_{Bs} is the set of SysML blocks that defines components,
3. Γ_{Bf} is the set of SysML blocks for flow ports typing,
4. Γ_{Enum} is the set of enumerations,
5. Γ_{Inter} is the set of interactions,
6. τ_a is the absolute time,
7. τ_s is the time step.

4.2.3/ Block Definition Diagram

A block definition diagram for Modelica simulation purpose may contain blocks and enumerations. A block is defined with attributes and may be composed of other components. Thus, it may have different typed elements (properties, parts and flow ports). A block's behaviour may be specified by constraints and state diagram, which transition's guard and effect are specified using a subset of the Modelica language.

Definition 2: Block for component definition

We define $\beta \in \Gamma_{Bs}$ to be the tuple $\beta =< \eta, \Gamma_{Att}, \Gamma_{Part}, \Gamma_{FP}, \Gamma_{Cnt}, \Gamma_{Cons}, \Gamma_{SM}, \tau_r >$, where:

1. η is the unique name of the block,
2. Γ_{Att} is the set of attributes,
3. Γ_{Part} is the set of parts,
4. Γ_{FP} is the set of flow ports,
5. Γ_{Cnt} is the set of connectors,
6. Γ_{Cons} is the set of constraints,
7. Γ_{SM} is the set of parallel state machines,
8. τ_r is the relative time of the components.

Each attribute, each part and each flow port shall be typed with primitive types (real, integer and Boolean, respectively noted $\mathbb{R}$, $\mathbb{Z}$ and $\mathbb{B}$) or with user-defined types (block for part, block for flow port typing, and enumeration, respectively noted Γ_{Bs}, Γ_{Bf} and Γ_{Enum}). The set of types Γ_T is defined by $\Gamma_T = \{\Gamma_{Bs}, \Gamma_{Bf}, \Gamma_{Enum}, \mathbb{R}, \mathbb{B}, \mathbb{Z}\}$.

Concerning attributes of Γ_{Att}, we have to distinguish several cases: an attribute may be a constant, an equation's unknown or a parameter. Within Modelica, an equation's unknown, which needs to be solved by integration, can be either continuous or discrete.

Definition 3: Attribute

Let $\alpha \in \Gamma_{Att}$ be defined by $\alpha =< \eta, \omega, \upsilon, t >$ where:

1. η is the name of the attribute,
2. ω is variability such as $\omega \in \{$constant, parameter, discrete, continuous$\}$,
3. υ is the value of the attribute,
4. t is the type of the attribute (Γ_{Enum}, $\mathbb{R}$, $\mathbb{B}$, or $\mathbb{Z}$).

If an attribute is discrete or continuous, then it is necessarily a state variable.

An enumeration is composed of enumeration literals. The following is the definition of such enumerations.

Definition 4: Enumeration

We define $\epsilon \in \Gamma_{Enum}$ such as $\epsilon = \{lit_1, lit_2, ..., lit_n\}$, where $lit_1, lit_2, ..., lit_n$ are enumeration literals.

4.2.4/ Internal Block Diagram

The IBD may contains parts with flow ports and connections between these ports. We formalize blocks that serve for flow port typing and the connection between flow ports.

A block for flow ports typing can only have properties, i.e., attributes that describe what flows between ports. Then, the following is the definition of such blocks:

Definition 5: Block for flow ports typing

We define $\beta_f \in \Gamma_{Bf}$ to be the tuple: $\beta_f =< \eta, \Gamma_{Att} >$, where η is the unique name of the block and Γ_{Att} is the non-empty set of attributes.

We need also to formalize the connection between parts of a SysML model. Connections are always between two flow ports, and a flow port has to be connected at least to one other flow port. Then, we define the surjective connecting function as follows:

Definition 6: Connecting function

Let f_c, the surjective connecting function, be defined by $f_c : \Gamma_{FP} \times \Gamma_{FP} \twoheadrightarrow \Gamma_{Cnt}$.

4.2.5/ State Machines and Continuous Behaviour

The continuous behaviour of the system is specified by equations over continuous state variables. The SysML constraints (Γ_{Cons}) are written using a subset of the Modelica language that expresses equations. This subset is presented in Fig. A.1 (see Appendix A). The order of appearance of equations is not important since they are all evaluated at each time step. Numerical solvers (embedded in all Modelica frameworks) are able to rewrite such constraints into a set of first-order differential equations in order to compute integration over time.

State machine diagrams enable to describe the life-cycle of a SysML block. For instance, one may specify several component states depending on time, state variables or user behaviours. The language $\mathcal{L}_s$, used for specifying transition's guards and effects, is a subset of the Modelica language. A transition's guard is a Modelica Boolean expression. The formal definition of such state machines is given below. This definition excludes join, fork and history pseudo-states that are not supported yet.

Definition 7: State machine

State machine for simulation is defined as $SM =< s_0, \Sigma, \Gamma_E, \mathcal{L}_s, \delta >$, where:

1. s_0 is the initial state,
2. Σ is a finite non-empty set of states composed of three disjoint sets: simple states Σ_{ss}, compound states Σ_{cs} and eventually final states Σ_{fs},
3. Γ_E is the set of trigger events,
4. $\mathcal{L}_s$ is the alphabet for specifying guard and effect of a transition,
5. $\delta : \Sigma \times \Gamma_E \times \mathcal{L}_s \rightarrow \Sigma$ is the transition function.

The Modelica subset for guard specification is presented in Fig. A.2 (Expression Section B.2.7 of the Modelica grammar specification [Association, 2012]). It has been cleaned up to support Boolean expressions only.

An effect is a Modelica statement such as *assignment*, *if-statement*, *while-statement* or *for-statement*. The grammar of Modelica statement is available in Fig. A.3. Moreover, each state may have $onEntry$ and $onExit$ actions, which are respectively executed at the entry and the exit of the state. These are defined using Modelica statements (see Fig. A.3). Concerning trigger events Γ_E, we only consider call events, i.e representing an operation call. The called operations have to be defined in the block that the state machine specifies. However, we do not consider the operations of the blocks for components definition because operations are not translated into Modelica code, only trigger events are.

4.2.6/ SEQUENCE DIAGRAM

The sequence diagram describes the flow of control between actors and systems (blocks) or between parts of a system. In the definition 1, we denoted the sequence of actions by the set Γ_{Inter}. Within our formal framework, we considered lifelines, asynchronous messages, action execution specifications, and duration constraints to specify actions duration. Note that a complete formalism of UML sequence diagram was proposed in [Li et al., 2004].

A lifeline represents necessarily an existing part or actor. Therefore, we can define a total injective function $f_l : \Gamma_{Ll} \rightarrowtail \Gamma_{Part}$ that associates a lifeline to a part. In our context, an asynchronous message specifies an action of a block. Then a message triggers an operation of a component, i.e. its signature is an operation.

Definition 8: Interactions

We define $\iota \in \Gamma_{Inter}$ to be the tuple: $\iota =< \eta, \Gamma_{Ll}, \Gamma_{AsMess}, \Gamma_{AcExSpec}, \Gamma_{DurCons} >$, where:

1. η is the unique name of the interaction,
2. Γ_{Ll} is the set of lifelines,
3. Γ_{AsMess} is the set of asynchronous messages,
4. $\Gamma_{AcExSpec}$ is the set of action execution specifications,
5. $\Gamma_{DurCons}$ is the set of duration constraints.

4.3/ RUNNING EXAMPLE

In this section, we illustrate the proposed SysML formal modelling framework for simulation purpose. The running example, presented in Sect. 1.5 is modeled with SysML. Then the SysML4Modelica profile is applied to perform simulations.

4.3.1/ THE SYSML CONTINUOUS MODEL

From the running example, we identify five blocks: the environment, the tank 1 with its associated controller, and the tank 2 with its associated controller. Figure 4.1 shows the SysML BDD of the running example, profiled with `SysML4Modelica` constructs. It, comprises 5 blocks: `TankSystem`, `Tank`, `Controller`, `Environment`, and `LiquidSource`.

Each SysML block for component definition is profiled with the `<<modelicaModel>>` stereotype. Concerning attributes, each of them is profiled with the `<<modelicaValueProperty>>` stereotype. It enables to specify which attribute is a parameter, a constant or a continuous variable. Table 4.3 summarizes the variability of each attribute.

Table 4.3: Variability of each Attribute

Attribute	Variability	Attribute	Variability
Tank.area	constant	ReadSignal.val	Continuous
Tank.flowGain	constant	Tank.h	Continuous
Tank.minV	constant	LiquidSource.isOn	Continuous
Tank.maxV	constant	LiquidSource.sourceValue	Continuous
Controller.K	constant	Controller.x	Continuous
Controller.T	constant	Controller.delta	Continuous
Controller.ref	Parameter	ActSignal.val	Continuous
LiquidFlow.val	Continuous		

We have specified 3 `<<modelicaConnector>>` blocks for flow port typing: `ActSignal`, `ReadSignal`, and `LiquidFlow`. `ActSignal` represents the controller's actuator signal, `ReadSignal` represents the controller's sensor signal, and `LiquidFlow` is the flow of liquid for the source.

The equations of the running example (see Sect. 1.5) are written as SysML constraints stereotyped by <<modelicaEquation>>. Note that according to Equation (10) and (11), s_{out} depends on time:

$$time < 150 \implies s_{out} = 0.02(m^3 \cdot s^{-1}) \tag{10}$$

$$time \geq 150 \implies s_{out} = 0.06(m^3 \cdot s^{-1}) \tag{11}$$

Presently, we consider that the liquid source provides constant flow: $s_{out} = 0.02$.

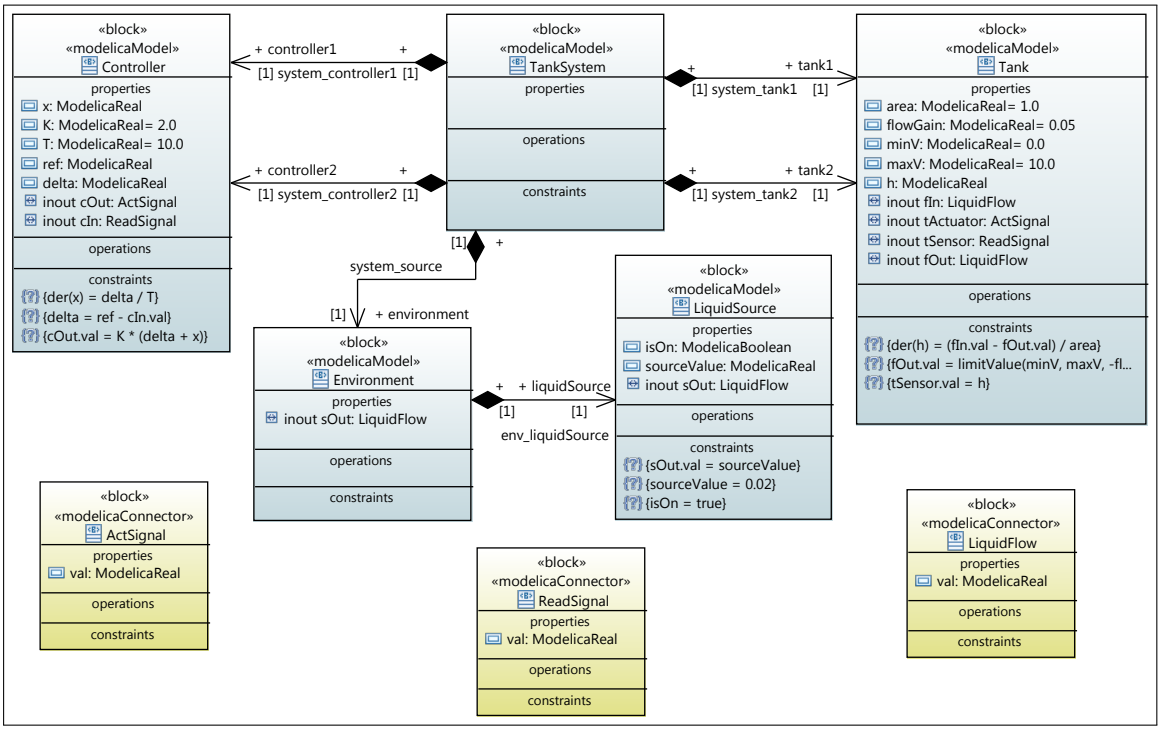


Figure 4.1: Tank System BDD

Then, each component is connected in the Internal Block Diagram as depicted in Fig. 4.2. Each part is profiled with the <<modelicaPart>> stereotype. This stereotype allows us to specify local modification over the controller 1 and the controller 2 by defining the parameter ref: $controller1.ref = 0.25$, $controller2.ref = 0.4$. Therefore, the liquid height h inside the tank1 has to be stabilized around 0.25, whereas the liquid height h inside the tank2 has to be stabilized around 0.4.

Finally, let say that one want to know the state of each tank, i.e. empty, partially filled, and full. Therefore, as illustrated in Fig. 4.3, we have specified a state machine inside the block `Tank`. The transitions are guarded with Boolean expression over the liquid height h.

The following is the list of the SysML constructs for continuous simulation regarding the formal framework defined in this chapter. Then the model for simulation is defined as Model $M_s =< TwoTankSystem, \Gamma_{Bs}, \Gamma_{Bf}, \Gamma_{Enum}, \Gamma_{Inter} >$, where:

- $\Gamma_{Bs} = \{TankSystem, Tank, Controller, Environment, LiquidSource\}$
- $\Gamma_{Bf} = \{ActSignal, ReadSignal, LiquidFlow\}$

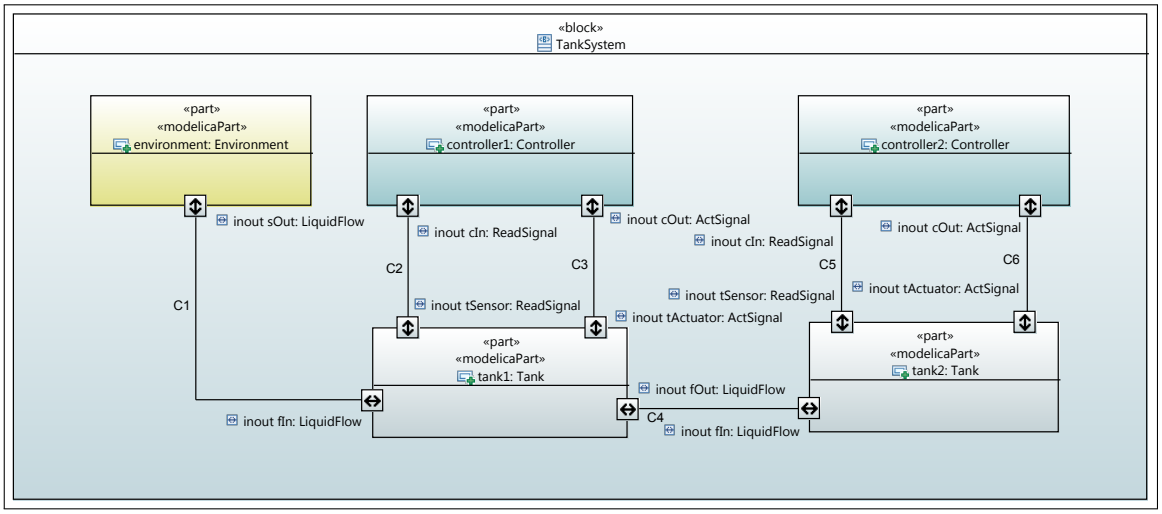


Figure 4.2: Tank System IBD

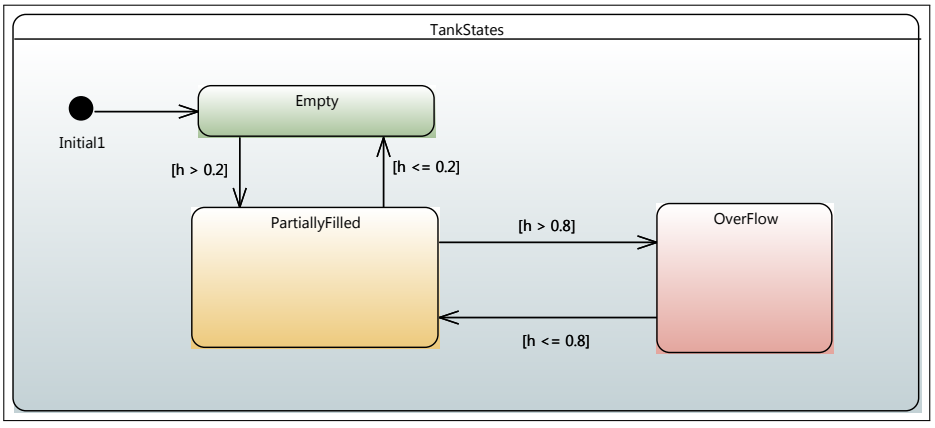


Figure 4.3: Tank State Machine

- $\Gamma_{Enum} = \emptyset$
- $\Gamma_{Inter} = \emptyset$

Note that $\Gamma_{Inter} = \emptyset$ as interactions are used to represent test scenario. We illustrate the use of sequence diagram in the next chapter (see Sect. 5.4.3). The block `Tank` contains attributes, constraints and a state machine. Therefore, $Tank =< \eta, \Gamma_{Att}, \Gamma_{Part}, \Gamma_{FP}, \Gamma_{Cnt}, \Gamma_{Cons}, \Gamma_{SM} >$, where:

- $\eta = Tank$,
- $\Gamma_{Att} = \{area, flowGain, minV, maxV, h\}$,
- $\Gamma_{Part} = \emptyset$,
- $\Gamma_{FP} = \{f_{in}, f_{out}, t_{actuator}, t_{sensor}\}$,
- $\Gamma_{Cnt} = \emptyset$,

- $\Gamma_{Cons} = \{C1, C2, C3\}$ and $C1 =$ "$der(h)...$", $C2 =$ "$fOut.val = ...$", and $C3 =$ "$tSensor.val = h$",
- $\Gamma_{SM} =< s_0, \Sigma, \Gamma_E, \mathcal{L}_s, \delta >$, where:
  - $s_0 = Initial1$,
  - $\Sigma = \{Empty, PartiallyFilled, Overflow\}$,
  - $\Gamma_E = \emptyset$,
  - $\mathcal{L}_s$ is the Modelica subset for specifying guards and effects,
  - δ is the transition function.

The next section presents the simulation of this system.

4.3.2/ SIMULATION RESULTS

The SysML model of the tank system was transformed into Modelica using model transformation and code generation techniques (see Chapter 6). The resulting Modelica code for the block `Tank` is available in Fig B.10. Each state of the state machine is transformed into Boolean variable. This enables to view the states of the tank. For instance, the Fig. 4.4 shows the liquid height and the states of both tanks. We see that the tank 2 is overflowing between $52s$ and $72s$. After $150s$, we observe that the liquid height is stabilized to the previously defined ref values: $controller1.ref = 0.25$ and $controller2.ref = 0.4$. Therefore, this first simulation permits to verify the functional requirements of the controller, i.e. the liveness property, and the stability property.

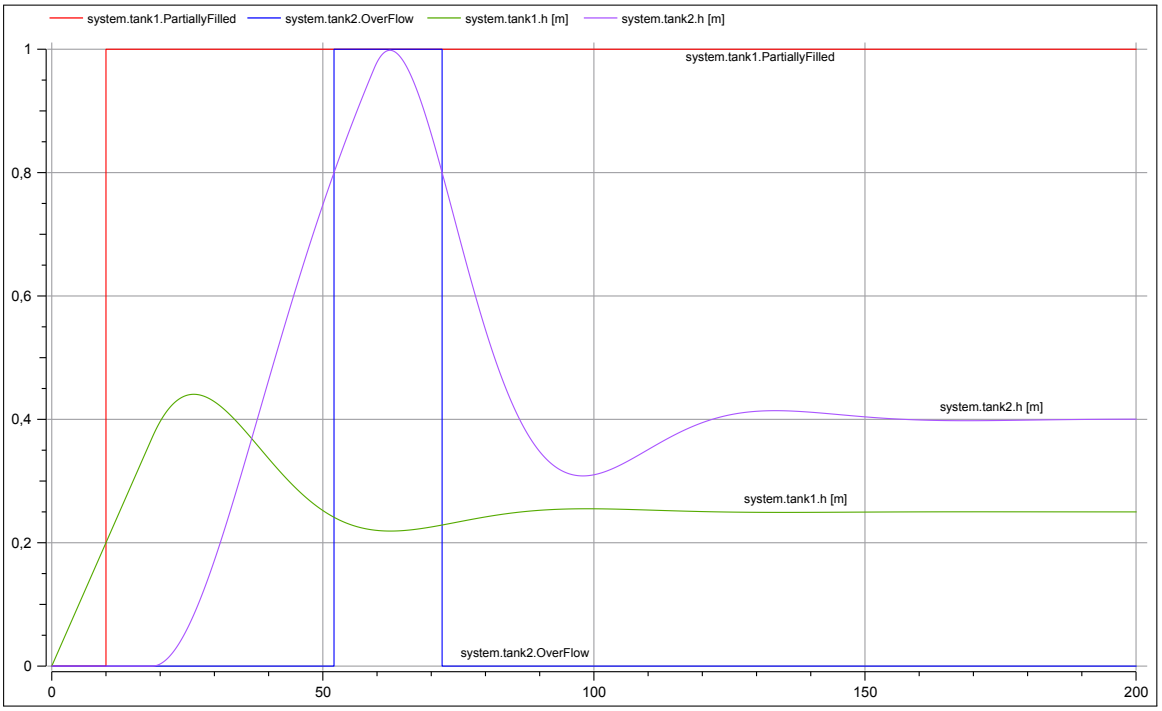


Figure 4.4: Simulation Results of the Liquid Height in Function of the Time

4.4/ ASSESSMENT

In this chapter we have presented the SysML subset used within the SysML4Modelica profile and its extended version for state machines and sequence diagram. This subset is sufficient to perform Modelica code generation and simulation. It enables to validate a system at the earliest stage of a design process by automating the derivation of Modelica code [Gauthier et al., 2015b].

The gap between SysML and Modelica being quite important at business-level, the cost to perform simulation from SysML model can be also important since design teams have to learn two languages. This could be somehow an issue for existing processes in the industry. Engineers that are familiar with the use of SysML should learn the basics of Modelica to correctly apply the SysML4Modelica profile. Nevertheless, from our experience, the efforts spent to learn the basics of Modelica and to apply the SysML4Modelica profile were weak compared to the time saved by automatically generating the simulation code. It should also be underlined that the SysML4Modelica profile can be applied to existing SysML models without changing the overall structure of the model. Incomplete model can also be handled: we could generate the structure of Modelica code without taking into account all the behavioral aspects (equations, algorithms, etc).

In addition, to describe complex and heterogeneous systems, the SysML4Modelica profile enables to bring together, in a single model, the non executable graphical high-level SysML modelling and the real-time and continuous Modelica specifications. However, no theoretical framework is given to provide a practical way to combine the architecture and discrete behaviours of SysML models with the continuous aspects described by Modelica formula. This thesis bridges this gap by defining such a framework to bring them back together to achieve model-based testing. It integrates constraint solving to address discrete animation and black-box test generation, and Modelica simulation to address continuous needs.

To provide a modelling framework that enables to perform both simulation and testing from a single SysML model, discrete aspects of the system have also to be integrated in order to use the CLPS-BZ solver for animation and test case generation purposes. This last, is the topic of the next chapter.

5

SysML MODELLING FOR ANIMATION AND TESTING

Contents

This chapter gives a formal description of the SysML modelling framework from the abstract and discrete point of view. We first introduce in Sect. 5.1 the CSP concepts and its relative constructs within the CLPS-BZ solver. Then, we present in Sect. 5.2 the formal modelling framework that enables to model abstract and discrete parts of a system for animation and test generation. Finally, we illustrate in Sect. 5.4 this framework with the running example given in Sect. 1.5.

5.1/ SysML FOR CLPS-BZ

The constraint system, described using the BZP format for CLPS-BZ, obviously defines a Constraint Satisfaction Problem (CSP) [Macworth, 1977], i.e. a set of constraints, which must be satisfied by the solution of the problem it models. Formally, a CSP is a triplet $< V, D, C >$ where V is a set of variables $\{v_1, \ldots, v_n\}$, D is a set of domains $\{d_1, \ldots, d_n\}$, where d_i is the domain associated with the variable v_i, and C is a set of constraints $\{c_1(V_1), \ldots, c_m(V_m)\}$, where a constraint c_j involves a subset V_j of the variables of V. Within CLPS-BZ, which is able to manage sets and integer finite domains, variables of V can be either an $atom$, or a set of atoms ($set(atom)$), or a (nested) set of (nested) pairs of atom ($set(pair(atom, atom))$). The SysML subset for animation enables to formally specify the system to perform a constraint evaluation.

The formal structures used in this chapter are the same as presented in Sect. 4.2.1. In the next section, we present the formal modelling framework for animation.

5.2/ FORMAL MODELLING FRAMEWORK

The SysML subset for animation purpose focuses on the following diagrams: BDD, IBD and state machine. The structural view of the system is specified in the BDD with blocks, which are connected each other using flow ports that are depicted in the IBD. The behaviour of each subsystem may be described using state machines or operations with OCL constraints on transitions or on operation precondition and postcondition.

5.2.1/ MODEL FOR ANIMATION

A SysML model for animation describes the system from an abstract and discrete point of view. The model is abstract in the way that the domain of a variable in $\mathbb{R}$ is discretized using enumeration classes since CLPS-BZ only manages integers, Booleans and finite sets. The behaviour of the model is also discrete as, during animation, we do not know what happen between two stable states of the state machines. Of course, simulation gives us some information about it, but during model animation, each state transition is executed as an atomic and non-breaking computation.

We define a SysML model for animation as a model M_a comprising blocks, enumerations, and associations. SysML enumerations enable declaring abstract types that may be used during a animation.

Definition 9: Model for animation

The model for animation M_a is defined by $M_a =< \eta, \Gamma_{Ba}, \Gamma_{Enum}, \Gamma_{Asso} >$, where:

1. η is the name of the model,
2. Γ_{Ba} is the set of SysML blocks that defines components,
3. Γ_{Enum} is the set of enumerations,
4. Γ_{Asso} is the set of associations between blocks.

5.2.2/ BLOCK DEFINITION DIAGRAM

A block definition diagram for CSP animation may contain blocks, enumerations, and associations.

A block for component definition comprises attributes, parts and operations, which are used to describe actions from the environment.

Definition 10: Block for component definition

We define $\beta \in \Gamma_{Ba}$ to be the tuple $\beta =< \eta, \Gamma_{Att}, \Gamma_{Part}, \Gamma_{Op}, \Gamma_{SM} >$, where:

1. η is the unique name of the block ,
2. Γ_{Att} is the set of attributes,
3. Γ_{Part} is the set of parts,
4. Γ_{Op} is the set of operations,
5. Γ_{SM} is the set of parallel state machines.

Concerning enumerations (see Definition 4) they are translated into $set(atom)$, where the atoms are the literals defined in the enumeration. Thus, enumerations define domains in the CSP.

Associations of Γ_{Asso} are translated into relations between instances of classes. The multiplicities of the association determine whether the relationship is a function, and if so, the type of this function (partial or total, and possibly injective, surjective or bijective). Moreover, role names are present at each end of an association.

Definition 11: Association

We define $\zeta \in \Gamma_{Asso}$ between two blocks β_a and β_b to be the tuple $\zeta =< \eta, \beta_a, \beta_b, m_a, m_b, r_a, r_b >$, where:

1. η is the unique name of the association,
2. β_a is the supplier block,
3. β_b is the client block,
4. m_a is the multiplicity of β_b in $beta_a$,
5. m_b is the multiplicity of β_a in $beta_b$,
6. r_a is the role name of β_b in relation to β_a,
7. r_b is the role name of β_a in relation to β_b.

Let's take the model of the Fig 5.1: A and B are two classes, $asso$ is an association between A and B. The multiplicities are ma and mb, role names are ra and rb. The following Table 5.1 summarizes the translation of an association to a relationship according to its multiplicities [Bruel, 1996]. In this Table, Γ_{A_I} and Γ_{B_I} denote respectively the set of instances of A and the set of instances of B.

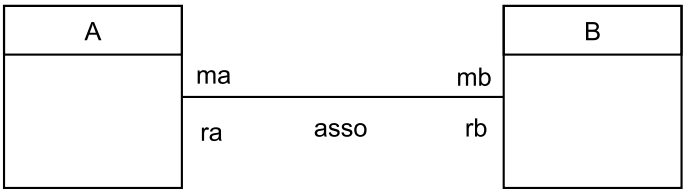


Figure 5.1: Association Example

Definition 12: Attribute

Let $\alpha \in \Gamma_{Att}$ be defined by $\alpha =< \eta, \omega, \upsilon, t >$ where:

1. η is the name of the attribute,
2. $\omega \in \{$constant, variable$\}$, and if $\omega = variable$ then ω is a state variable,
3. υ is the value of the attribute during the animation,
4. t is the type of the attribute ($\mathbb{Z}$, $\mathbb{B}$ or Γ_{Enum}).

Each attribute $\alpha \in \Gamma_{Att}$, belonging to a block β, is translated into a total function between all instances of the block β and the domain of α. Considering for example that α is an integer, α is translated into a structure of type $\Gamma_{Part} \times \mathbb{Z}$.

For animation, the set of types Γ_{Ta} is defined by $\Gamma_{Ta} = \{\Gamma_{Ba}, \Gamma_{Enum}, \mathbb{B}, \mathbb{Z}\}$.

Table 5.1: Mapping Between Association and Relationship

Multiplicity ma **of** a	**Multiplicity** mb **of** b	**Relationship**
1..*	1..*	Relation $\Gamma_{A_I} \leftrightarrow \Gamma_{B_I}$
0..*	1..*	Relation $\Gamma_{A_I} \leftrightarrow \Gamma_{B_I}$
0..*	0..*	Relation $\Gamma_{A_I} \leftrightarrow \Gamma_{B_I}$
Generalization of the above cases ($a \geq 0$, $b \geq a$, $b > 1$, $c \geq 0$, $d \geq c$, $d > 1$)		
a..b	c..d	Relation $\Gamma_{A_I} \leftrightarrow \Gamma_{B_I}$
1..*	0..1	Partial surjective function $\Gamma_{A_I} \nrightarrow \Gamma_{B_I}$
0..1	1..*	Partial surjective function $\Gamma_{B_I} \nrightarrow \Gamma_{A_I}$
Generalization of the above cases ($a \geq 1$ and $b > a$)		
a..b	0..1	Partial surjective function $\Gamma_{A_I} \nrightarrow \Gamma_{B_I}$
0..1	a..b	Partial surjective function $\Gamma_{B_I} \nrightarrow \Gamma_{A_I}$
1..*	1	Total surjective function $\Gamma_{A_I} \twoheadrightarrow \Gamma_{B_I}$
1	1..*	Total surjective function $\Gamma_{B_I} \twoheadrightarrow \Gamma_{A_I}$
Generalization of the above cases ($a \geq 1$ and $b > a$)		
a..b	1	Total surjective function $\Gamma_{A_I} \twoheadrightarrow \Gamma_{B_I}$
1	a..b	Total surjective function $\Gamma_{B_I} \twoheadrightarrow \Gamma_{A_I}$
0..1	1	Total injective function $\Gamma_{A_I} \rightarrowtail \Gamma_{B_I}$
1	0..1	Total injective function $\Gamma_{B_I} \rightarrowtail \Gamma_{A_I}$
0..1	0..1	Partial injective function $\Gamma_{A_I} \nrightarrow \Gamma_{B_I}$
1	1	Bijective total function $\Gamma_{A_I} \rightarrowtail \Gamma_{B_I}$
0..*	1	Total function $\Gamma_{A_I} \rightarrow \Gamma_{B_I}$
1	0..*	Total function $\Gamma_{B_I} \rightarrow \Gamma_{A_I}$
Generalization of the above cases ($b > 1$)		
0..b	1	Total function $\Gamma_{A_I} \rightarrow \Gamma_{B_I}$
0..*	0..1	Partial function $\Gamma_{A_I} \nrightarrow \Gamma_{B_I}$
0..1	0..*	Partial function $\Gamma_{B_I} \nrightarrow \Gamma_{A_I}$
Generalization of the above cases ($b > 1$)		
0..b	0..1	Partial function $\Gamma_{A_I} \nrightarrow \Gamma_{B_I}$

The operations have a name and optional parameters (in, out, $inout$, $return$). For animation purpose, we only take into account in and $return$ parameters. In addition, operations can also have OCL4MBT precondition and postcondition.

Definition 13: Operation

Let $o \in \Gamma_{Op}$ defined as $o =< \eta, \Gamma_{Par}, pre, post >$ where: η is the name of the attribute, Γ_{Par} is the set of parameters, pre is the precondition of the operation and $post$ is the postcondition of the operation.

There are two possible operational interpretations of OCL4MBT. First, the passive context is used to specify operation preconditions. Passive OCL expressions check the state variables of a model, i.e., they do not modify the model state.

Second, the active context is used to specify operation postconditions. Active OCL expressions change the values of state variables and define values for the return parameter of operations. Operation parameters are translated into $atom$ and operations define constraints of the CSP by their preconditions and postconditions.

5.2.3/ INTERNAL BLOCK DIAGRAM

The IBD may contains parts with flow ports and connections between these ports. We formalize here parts that serve for instance creation within the CSP.

Blocks define variables of the CSP and their domains are defined by the set of instances Γ_{Part} of these blocks. With CLPS-BZ, each block is associated with information concerning its instances: the set of instances that can potentially be created ($all\_instances$ as a $set(atom)$), the set of currently created instances ($instances$ as a $set(atom)$), and the current instance, which is the last created instance or treated by an operation ($currentInstance$ as an $atom$). Among all the possible instances $all\_instances$, a fictitious $none$ instance is created. It is used to formalize the absence of current instance.

For CSP solving, flow ports and connections are not supported yet since abstract and discrete animation does not deal with signal event of energy flowing. However, these elements play a major role for continuous simulation.

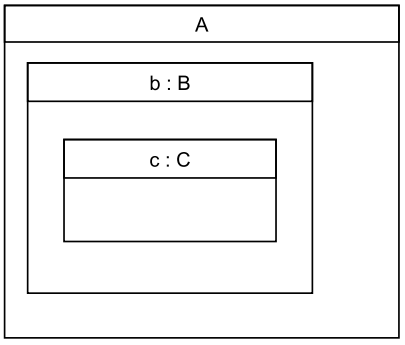


Figure 5.2: Nested Parts and IBD

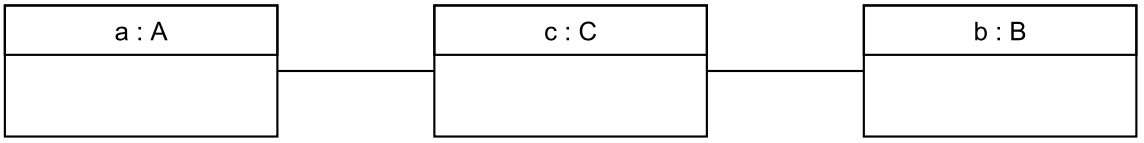


Figure 5.3: Flattened Instances and Links Using Parts and Associations

One major issue during the creation of the set of instances is the creation of the links between these instances. Although dynamic link creation and dynamic link destruction are not considered for SysML model (a component can not disappear from a system), we have had to take into account links regarding the associations they refer to. For instance, let a component C as a part of a component B, B being also a part of a given component A as shown in Fig. 5.2.

Then we have had to create the links as illustrated in Fig. 5.3. Note that the block A was not initially instantiated. Therefore, we also have had to consider two cases such that the restriction saying that, each block of Γ_{Ba} must be instantiated at least once, is always satisfied. Either it exists a part ρ in Γ_{Part} such that $\rho.type \in \Gamma_{Ba}$, then $\rho \in all\_instances$. Or it does not exist a part ρ in Γ_{Part} and in this case we need to create one during the translation to CSP.

5.2.4/ STATE MACHINES

State machines are used to specify discrete component behaviours and external, physical or human, actions. For animation purpose, state machines are defined as expressed in the definition 7. However, we define $\mathcal{L}_a$, based on the OCL subset of the Fig. A.4 and A.5 (see Appendix A), as the alphabet for specifying guard and effect of a transition.

Each state (single, composite, initial or final state) of a state machine is translated into a specific context of the CSP. For each state, a variable $status$ stores the current state(s) of a block instance: it is a function associating each instance of the block to a Boolean (the function is partial due to the presence of the fictitious $none$ instance in its domain). At the beginning of the animation, each instance is in the initial state. In addition, two operations are declared to each state to formalize the possible $onEntry$ and $onExit$ effects.

Each state machine is associated with the block it specifies the behaviour. Operations of this block can be used as triggers for some transitions of the state machine. To avoid unmanageable infinite loop during animation, three types of transitions are allowed: external (reflexive or not) with trigger, internal with trigger and guarded external or automatic (not reflexive). Therefore, automatic or guarded internal transitions, external guarded or automatic reflexive transitions, and trans-hierarchical transitions are forbidden.

Consider the following example of the Fig. 5.4 where $op1$ is an operation with a precondition and a postcondition:

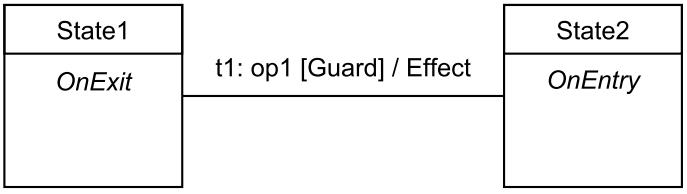


Figure 5.4: Transition Example

The execution order of the various elements defining the transition t1 is:

1. precondition of the triggered operation $op1$,
2. postcondition of the triggered operation $op1$,
3. Guard of transition $t1$
4. $OnExit$ the state $State1$
5. Effect of transition $t1$
6. OnEntry of the state $State2$.

To respect this order, information relating to the transition are translated into an `event` predicate. However, if an operation is executed and if one is in a state from which a transition can be triggered by this operation, then it is essential to fire this transition just after the execution of the operation. Note also that the guard and the effect of the transition may use values of the parameters of the triggered operation. We have to keep these values between the execution of the operation and the execution of the transition.

For this reason, in the BZP file and in the context of the class containing the state machine, we add the following elements. First, a set $set\_operations\_triggers = \Gamma_E$ containing all operations used as a trigger for a transition in the state machine is declared (as well as a *none* element to specify the fact that no operations have been triggered). Second, a variable $opCalled \in \Gamma_E$, declared to store the last executed operation, enables to fire, from the current state, transition triggered by this operation.

We finally add a precondition for all guarded and automatic transitions, expressing that no operation has been called ($opCalled = none$). This ensures that the UML "run-to-completion" semantic is satisfied. The "run to completion" event is initiated only after the execution of an transition event inducing a state change in order to reach a stable state. For now, the only executable transitions during a "run-to-completion" are guarded or automatic transitions. Indeed, triggered operations are not permitted in the postcondition of operations or in the effect of transitions since there is no new event during a "run-to-completion".

To translate this "run-to-completion" we define a Boolean variable $runToCompletion$ in the BZP file. This variable is assigned to the true value only in the effect of external event transitions. All automatic and guarded transitions have the $runToCompletion$ variable assigned to true in their precondition. Finally, $runToCompletion$ must be assigned to false if a stable state is reached. The state machine is in a stable state since no automatic or guarded transitions can be fired.

To sum up, each state gives rise to a variable $status$ and constraints related to the $onEntry$ and $onExit$ actions. Each transition is translated into constraints in the CSP that are defined by its guard and effect. Trigger events of Γ_E define a set of operation triggers $set(atom)$ that defines the domain of the variable $opCalled$. Finally, a $runToCompletion$ variable is defined to ensure that the UML "run-to-completion" semantic is satisfied.

5.3/ COMBINED FORMALISM FOR SIMULATION AND ANIMATION

The above formalized subset enables to animate a discrete and abstract SysML model by translating it into a CSP. This CSP is defined by a Prolog-readable BZP file. Thus, it is now possible to specify the continuous and discrete behaviour of a complex and critical system for simulation, animation and testing purpose.

Table 5.2 summarizes the SysML subsets for simulation and animation. Each combined SysML element are derived both to Modelica element and to CSP element (variable V, domain D or constraint C). To propose a unified modelling framework, blocks and enumerations for simulation have to be used for animation. Then, the model for validation M_v is defined as $M_v = M_s \cap M_a = \{\Gamma_{Bv}, \Gamma_{Enum}\}$ where $\Gamma_{Bv} = \Gamma_{Bs} \cap \Gamma_{Ba}$. Blocks for flow port typing (Γ_{Bf}) are not used for animation.

Table 5.2: SysML for Modelica Simulation and CSP Animation

SysML elements	**Modelica elements**	**CSP <V,D,C>**
Model M_v	Root Modelica model	CSP model
Blocks Γ_{Bv}	Models	$\Gamma_{Bv} \in V$
Blocks Γ_{Bf}	Connectors	-
Enumerations Γ_{Enum}	Enumerations	$\Gamma_{Enum} \in D$
Attributes Γ_{Att}	Value properties	$\Gamma_{Att} \in V$
Constraints Γ_{Cons}	Equations	-
Parts Γ_{Part}	Components	$\Gamma_{Part} \in D$
FlowPorts Γ_{FP}	Ports	-
Connectors	Connect equations	-
Op. Precondition $pre \in \Gamma_{Op}$	Boolean expr ($\mathcal{L}_s$)	$pre \in C$ ($\mathcal{L}_a$)
Op. Postcondition $post \in \Gamma_{Op}$	Statement ($\mathcal{L}_s$)	$post \in C$ ($\mathcal{L}_a$)
Op. parameters Γ_{Param}	-	$\Gamma_{Param} \in D$
State-Machines SM	Algorithm sections	-
States Σ	Boolean variables	$status$ variable $\in V$
State $Entry$	Statement ($\mathcal{L}_s$)	$Entry \in C$ ($\mathcal{L}_a$)
State $Exit$	Statement ($\mathcal{L}_s$)	$Exit \in C$ ($\mathcal{L}_a$)
Event triggers Γ_E	Boolean variables	$\Gamma_E \in D$
Transitions δ	When statements	-
Transition $guard$	Boolean expr ($\mathcal{L}_s$)	$guard \in C$ ($\mathcal{L}_a$)
Transition $effect$	Statement ($\mathcal{L}_s$)	$effect \in C$ ($\mathcal{L}_a$)
Interactions Γ_{Inter}	Models	-
Lifelines Γ_{Ll}	Components	-
Messages Γ_{AsMess}	Boolean allocations	-
Duration constraints $\Gamma_{DurCons}$	Boolean expressions	-

Considering now $\beta_1 \in \Gamma_{Bs}$ and $\beta_2 \in \Gamma_{Ba}$, then $\beta_1 \cap \beta_2 =< \eta, \Gamma_{Att}, \Gamma_{Part}, \Gamma_{SM} >$ where each attribute of Γ_{Att} is defined as proposed in definition 12. Indeed, Modelica is able to process discrete and continuous variables whereas the CLPS-BZ solver is not able to manage continuous state variables. Concerning SysML parts, they enable to instantiate Modelica components and to declare block instances in the constraint system.

Finally, state machines for simulation and animation are not totally combined. The language for specifying guard and effect of transitions, as well as $onEntry$ and $onExit$ actions of states and $pre/post$ of operations, is indeed not fully equivalent. In one case, the language $\mathcal{L}_s$ is a subset of Modelica and in the other case, the language $\mathcal{L}_a$ is a subset of OCL. However, states, transitions and events are used for both simulation and animation. Thus, every state machines are translated both into Modelica code (using formula of $\mathcal{L}_s$) and CSP (using OCL code of $\mathcal{L}_a$). It should be noted that state machines without OCL code and event trigger are not translated into CSP because it would not impact the CSP solving and could even give a under-constrained CSP (and make it non deterministic).

5.4/ RUNNING EXAMPLE

In this section we illustrate the proposed formalism for animation with the running example. We first describe the changes made to the model of the Fig. 4.1. Then, the animation and test generation with the CLPS-BZ solver are presented. Finally, we show the concretization of a test sequence and its execution on the simulation model.

5.4.1/ THE SYSML MODEL FOR ANIMATION

Figure. 5.5 depicts the modifications on the model for animation purpose. As a reminder, Equations (10) and (11) rule the liquid source. We decided to specify such behaviours using a state machine over the flow level of the liquid source.

$$time < 150 \implies s_{out} = 0.02(m^3 \cdot s^{-1}) \tag{10}$$

$$time \geq 150 \implies s_{out} = 0.06(m^3 \cdot s^{-1}) \tag{11}$$

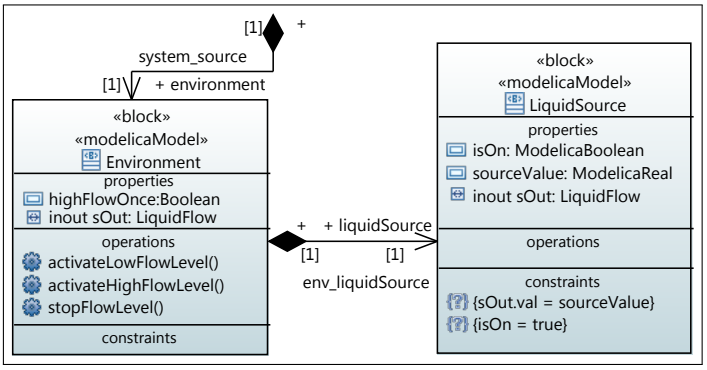


Figure 5.5: Modification of the BDD for CSP Solving

Considering that the liquid flow changes over time, and to illustrate discrete animation, we specified three operations. The first operation `activateLowFlowLevel()` enables to activate the liquid source with its smaller value $s_{out} = 0.02$. Then, the operation `activateHighFlowLevel()` enables to raise the liquid flow to a higher value $s_{out} = 0.02$, and the operation `stopFlowLevel()` permits to stop the flow of the liquid source. These operations are used as triggers in the state machine of the Fig. 5.6. We specified an OCL guard on the transition tr_low_no. It verifies that the property $highFlowOnce$ is true. This property is set to true in the effect of the transition tr_low_high. Thus, it ensures that the environment is at least in the state $HighFlowState$ before getting through the transition tr_low_no. Note that the transition effects may be specified both with Modelica code and OCL code (Modelica code is not used during animation, only OCL is).

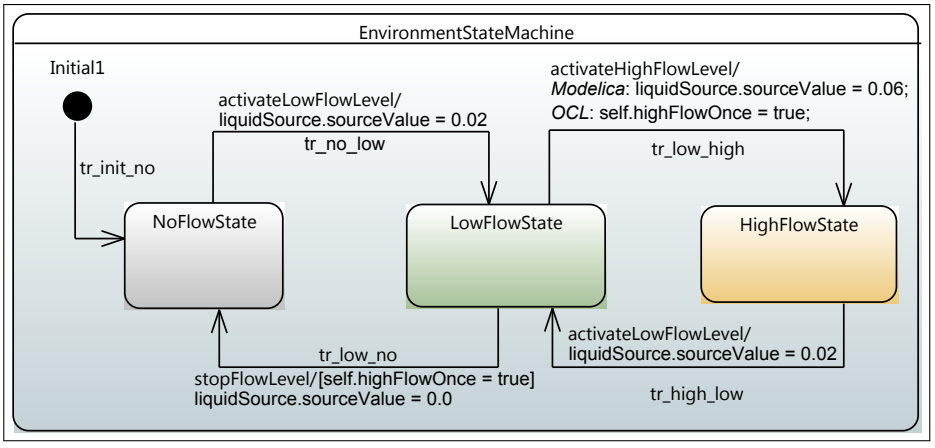


Figure 5.6: State Machine of the Environment

The model for animation is defined by $M_a =< \eta, \Gamma_{Ba}, \Gamma_{Enum}, \Gamma_{Asso} >$, where:

1. $\eta = TwoTankSystem$,
2. $\Gamma_{Ba} = \{TankSystem, Tank, Controller, Environment, LiquidSource\}$,
3. $\Gamma_{Enum} = \emptyset$,
4. $\Gamma_{Asso} = \{system\_tank1, system\_tank2, system\_controller1, system\_controller2, system\_source, env\_liquidSource\}$.

The block `Environment` contains a part, operations, and a state machine. Therefore, $Environment =< \eta, \Gamma_{Att}, \Gamma_{Part}, \Gamma_{Op}, \Gamma_{SM} >$, where:

1. $\eta = Environment$,
2. $\Gamma_{Att} = \emptyset$,
3. $\Gamma_{Part} = \{liquidSource\}$,
4. $\Gamma_{Op} = \{activateLowFlowLevel(), activateHighFlowLevel(), stopFlowLevel()\}$,
5. $\Gamma_{SM} = \{EnvironmentStateMachine\}$, and
   $EnvironmentStateMachine =< s_0, \Sigma, \Gamma_E, \mathcal{L}_s, \delta >$ where:
   - $s_0 = Initial1$,
   - $\Sigma = \{NoFlowState, LowFlowState, HighFlowState\}$,
   - $\Gamma_E = \{activateLowFlowLevel, activateHighFlowLevel, stopFlowLevel\}$,
   - $\mathcal{L}_a$ is the OCL subset for specifying guards and effects,
   - δ is the transition function.

5.4.2/ ANIMATION AND TEST GENERATION

The resulting BZP constraints of the `Environment` block, its state machine, the state `LowFlowState`, and the transition `tr_low_high` are shown respectively in Figures B.13, B.14, B.15, and B.16 (see Appendix B).

From the state machine of the Fig. 5.6, the solver is able to perform animation of the model. A complete animation sequence of the state machine is as follows:

$Initial1.onEntry \rightarrow tr\_init\_no.guard \rightarrow Initial1.onExit \rightarrow tr\_init\_no.effect \rightarrow NoFlowState.onEntry \rightarrow activateLowFlowLevel.pre \rightarrow activateLowFlowLevel.post \rightarrow tr\_no\_low.guard \rightarrow NoFlowState.onExit \rightarrow tr\_no\_low.effect \rightarrow LowFlowState.onEntry \rightarrow activateHighFlowLevel.pre \rightarrow activateHighFlowLevel.post \rightarrow tr\_low\_high.guard \rightarrow LowFlowState.onExit \rightarrow tr\_low\_high.effect \rightarrow HighFlowState.onEntry \rightarrow activateLowFlowLevel.pre \rightarrow activateLowFlowLevel.post \rightarrow tr\_high\_low.guard \rightarrow HighFlowState.onExit \rightarrow tr\_high\_low.effect \rightarrow LowFlowState.onEntry \rightarrow stopFlowLevel.pre \rightarrow stopFlowLevel.post \rightarrow tr\_low\_no.guard \rightarrow LowFlowState.onExit \rightarrow tr\_low\_no.effect \rightarrow NoFlowState.onEntry$

When no guard is specified, then it is considered as $true$. As depicted in Fig. B.11, the state machine is also translated into Modelica code in order to be simulated using the generated test cases. This Figure also illustrate the order of the statements if $onEntry$, $onExit$, pre, and $post$ were specified.

Finally, the CLPS-BZ solver has generated 18 test cases (from the animation) to cover the state $onEntry$, the state $onExit$, the transitions, and the operations. Note that covering the state $onEntry$ and the state $onExit$ is equivalent to state coverage.

The test cases may be concatenated to build test sequences. A simple test sequence that covers all states and all transitions is as follows:

$Initial1 \rightarrow NoFlowState \rightarrow activateLowFlowLevel() \rightarrow LowFlowState \rightarrow activateHighFlowLevel() \rightarrow HighFlowState \rightarrow activateLowFlowLevel() \rightarrow LowFlowState \rightarrow stopFlowLevel() \rightarrow NoFlowState$

The concretization and the execution of this test sequence are presented in the next section.

5.4.3/ CONCRETIZATION AND EXECUTION

From the test sequence presented in the previous section, we can build the sequence diagram of the Fig. 5.7. Sequence diagrams could be automatically designed from test sequences. In this thesis we manually created sequence diagrams from test sequences. This concretization needs human interventions to specify action durations.

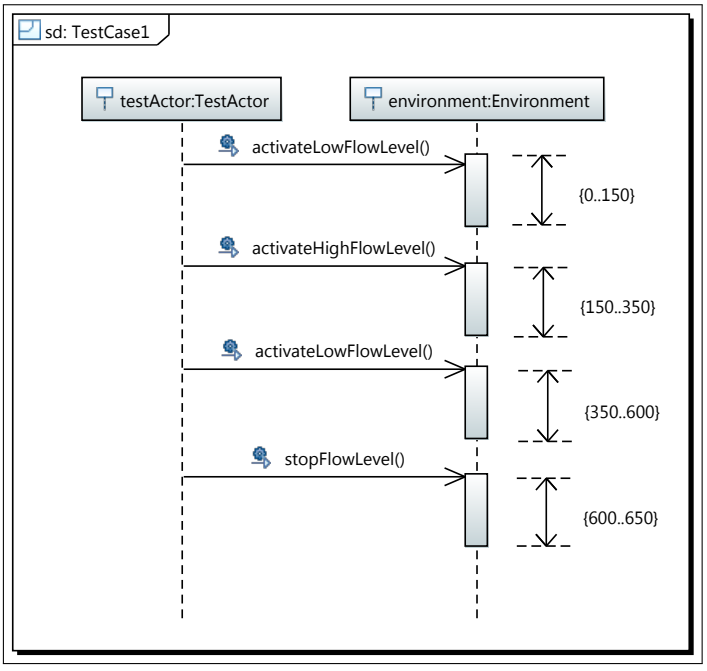


Figure 5.7: Test Sequence Generated by CLPS-BZ

Then, this sequence diagram is automatically transformed into Modelica code. The resulting Modelica code of this test sequence is available in Fig. B.12. Note that duration constraints serve to trigger `when` statements. Therefore, the sequence diagram, transformed into Modelica, enables to simulate the state machine of the environment in function of the time.

Finally, the simulation results are depicted in Fig. 5.8. It shows the level of liquid in each tank and the liquid flow of the liquid source component. The liquid flow level is a squared signal. Indeed, the liquid source value change instantaneously and remains constant for a period of time. It is an approximation of what happens in the reality. However, it could be possible to obtain more realistic test sequences by adding equations over the variations of the liquid flow level. This implies minor changes over the model, i.e. the transition effects of the environment state machine may be refined from $sourceValue = x$ to $sourceValue = f(t)$, where $f(t)$ is a function of time.

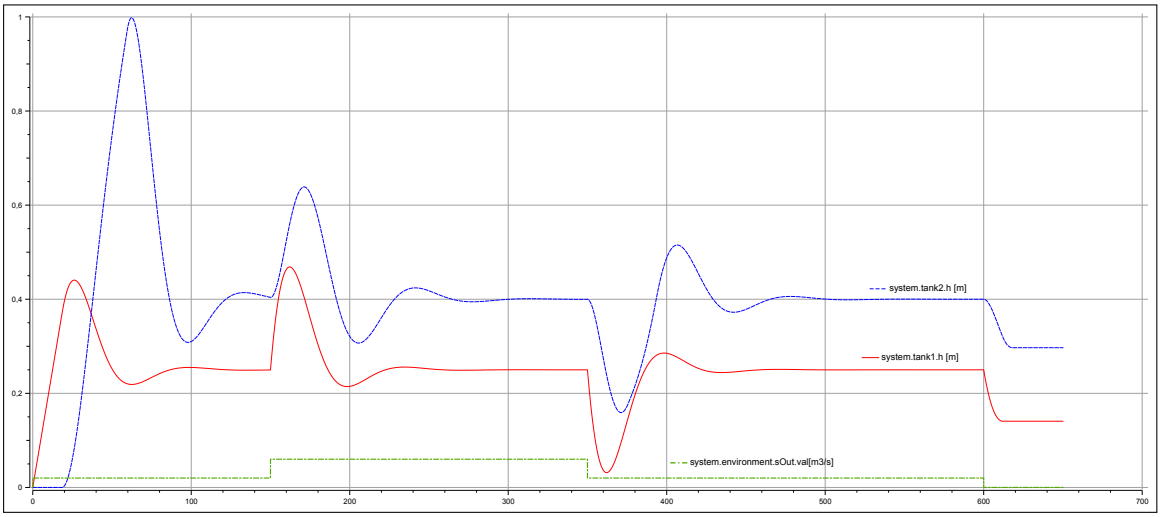


Figure 5.8: Execution of the Test Sequence on the Two Tanks System

5.5/ ASSESSMENT

This chapter presented a SysML framework that combines discrete aspects for model-based testing with continuous features for simulation. We formally described the SysML subsets CSP solving, and the way to combine it with the SysML subset for Modelica simulation in a single SysML model. Then, it is now possible to generate scenarios and initial conditions for simulation. This combined approach aims to be used within model-in-the-loop and hardware-in-the-loop processes.

In these processes, the simulation respectively plays two key roles: simulating a component based system and providing test cases and oracles for the model and its concrete product. While preserving the V-cycle to address complex and critical system development, we promote a more iterative and incremental approach driven by the early validation and verification activities. In addition, the concretization step is highly simplified for two reasons. First, we are able to specify guards and effects of transitions using a subset of the Modelica language. Therefore, using the proposed translation from SysML to Modelica enables to obtain an executable state machine pending events from the environment. Second, the sole human interaction is the duration constraints specification for the action.

III

Development and Experiments

6

PRACTICAL FRAMEWORK

Contents

Model transformation and code generation are the backbone of the Model Driven Architecture (MDA) approach [Group, 2003]. In the context of MBSE, this approach helps to bring the analysis of specifications and the rapid prototyping closer. Considering that SysML enables system modelling from specifications, MDA offers techniques to obtain executable Modelica prototypes from SysML models. In the same time, we propose to use MDA approaches to generate constraints for the CLPS-BZ solver.

In Sect. 6.1, we first present the process of Modelica code generation from SysML model. Then, we present the process of BZP constraints generation from SysML model in Sect. 6.2. Finally, we introduce the implementation in Sect. 6.3.

6.1/ FROM SYSML MODELS TO MODELICA CODE

The starting point of this translation process, depicted in Fig. 6.1, consists of giving Modelica semantics to SysML models using SysML4Modelica constructs. After verifying that the SysML model contains the correct Modelica constructs, a model transformation, based on the ATLAS Transformation Language (ATL) framework [Bézivin et al., 2003a, Bézivin et al., 2003b], is performed from the SysML4Modelica meta-model to the Modelica meta-model.

ATL is a model transformation language inspired by the OMG standard QVT[1]. It makes it possible to implement model transformation rules and to run transformation process. ATL rules are the heart of transformations since they describe how output elements (that conform to the output meta-model) are produced from input elements (that conform to the input meta-model). The ATL language is based on the OMG OCL (Object Constraint Language) for both its data types and its declarative expressions.

[1] http://www.omg.org/spec/QVT/1.1/

ATL provides three kind of rules namely *matched rule*, *called rule*, and *lazy rule*. The ATL matched rules allow to specify which source model element must be matched, the number and the type of the generated model elements and the way these target model elements must be initialized from the matched source elements. Contrary to matched rules, called rules enable to generate target model element from imperative code. This kind of rule must be called from an ATL imperative block. In addition, a called rule can accept parameters. Finally, lazy rules can be called from matched rules.

ATL allows to write methods with parameters and return type. These ATL functions are called `helpers`. They make it possible to define factorized ATL code.

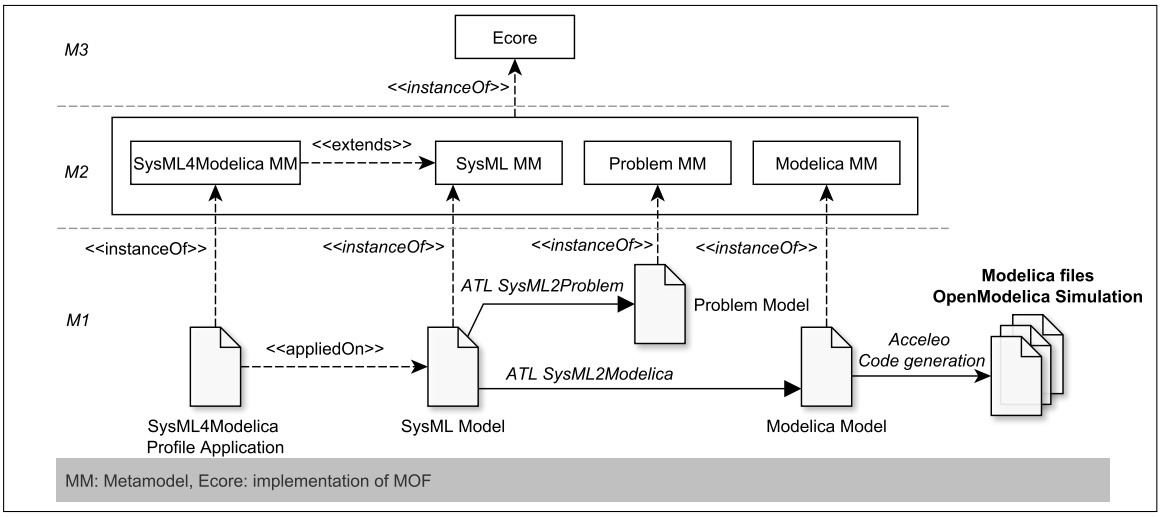


Figure 6.1: Modelica Generation Process from SysML Models

The last step of the proposed MDA approach concerns the Modelica code generation using the Acceleo technology[2]. Acceleo, developed by the company Obeo, is an open source code generator from the Eclipse foundation. It implements the MDA approach to develop application from EMF (Eclipse modelling Framework) based models. The Acceleo language is an implementation of the MOF Models to Text Transformation (MOFM2T) standard [OMG, 2008].

6.1.1/ SysML to Problem transformation

To ensure a consistent simulation and to avoid errors in the generated Modelica code, we need to verify SysML models before the translation to Modelica. The `SysML2Problem` verification is performed using ATL transformation rules. In order to generate problems, it is first necessary to build a meta-model that represents a so called problem. It is obvious that a problem should be clear enough to be corrected without loosing lot of time. Therefore, as depicted in Fig. 6.2, problem is described with its location and its nature (warning, error, critic).

[2]http://www.eclipse.org/acceleo/

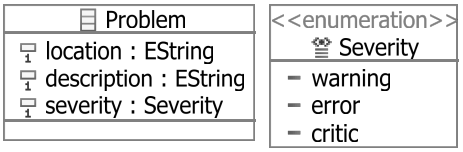


Figure 6.2: Problem Meta-model

Then, we have written ATL rules that implement the constraints defined in the OMG SysML-Modelica Transformation specification. For instance, considering a SysML block stereotyped with `<<modelicaModel>>`, the following constraint, has to be satisfied:

> Constraint 1: *A `modelicaModel` can only have Properties that are stereotyped by `modelicaPart`, `modelicaPort`, or `modelicaValueProperty`.*

For each SysML block stereotyped with `<<modelicaModel>>`, we need to verify whether the constraint is satisfied or not. Concretely, if the constraint is not satisfied, a problem is generated by the ATL rule of the Fig. 6.3. The `from` section of the rule is written as a pre-condition, which means that it has Boolean semantic. If the `from` section is true, a problem is automatically generated via the `to` section of the rule. Note that we check the applied stereotype with helpers. For instance, the `isModelicaModelStereotype()` helper returns true if the owner of the property, *i.e.* a block, is stereotyped with `<<modelicaModel>>`. The `SysML2Problem` transformation contains 34 ATL matched rules and 33 ATL helpers to perform the verification. The set of rules is available on a Github[3] repository.

```
1 helper context MMuml!NamedElement def: isModelicaModelStereotyped() : Boolean =
2 self.getAppliedStereotypes()->exists(s | s.qualifiedName = 'SysML4Modelica::Classes::
    ModelicaModel');
3
4 rule propertiesWellStereotyped{
5  from sysmlProperty: MMuml!Property(
6   sysmlProperty.owner.oclIsTypeOf(MMsysml!Block) and
7   sysmlProperty.owner.isModelicaModelStereotyped() and
8   not sysmlProperty.isModelicaPartStereotyped() and
9   not sysmlPropery.isModelicaValuePropertyStereotyped() and
10  not sysmlPropery.isModelicaPortStereotyped()
11 )
12 to
13  problem: MMproblem!Problem(
14   severity <- #error,
15   description <- 'The block '+sysml.owner.name+' can only have Properties that are
    stereotyped to modelicaPart, modelicaPort, or modelicaValueProperty',
16   location <- sysmlProperty.getQualifiedName()
17  )
}
```

Figure 6.3: ATL Implementation of the Constraint 1

Each problem is then shown to the modelling environment user interface. The engineer can brought corrections before the translation into Modelica. When no error is detected during the consistency verification, the SysML model is transformed into Modelica model. This transformation is the topic of the next section.

[3] https://github.com/SysMLModelicaIntegration/edu.ufc.femtost.disc.sysml2modelica

6.1.2/ SysML model to Modelica model

The SysML2Modelica ATL transformation aims to transform SysML models into Modelica models that are conformed to the Modelica meta-model depicted in Fig B.1. Finally, the generated Modelica model defines the entry point of Modelica code generation.

The proposed Modelica meta-model enables to represent the main Modelica objects. We do not represent here the full abstract syntax of Modelica. Indeed, EquationStatement and AlgorithmSection hold a body attribute typed with String to receive Modelica statements. This meta-model has been built with the Eclipse modelling Framework (EMF) as an ecore file. To perform the SysML2Modelica transformation, we have implemented 35 ATL matched rules, 2 called rules, 2 lazy rules, and 74 helpers.

6.1.3/ Modelica code generation

To perform code generation from models, we have used the Acceleo technology. As shown in Figure 6.4, this language uses an approach based on templates, which can be seen as a piece of code that creates reserved namespaces containing expressions on entities.

```
1  [template public generateModel(inModel:Model)]
2   [file [inModel.name]+'.mo', false, 'UTF-8')]
3    model [inModel.name/]
4    [for(inComponent : Component | inModel.modelicaComponents)]
5     [generateComponent(inComponent)/]
6    [/for]
7    [if(not inModel.equationSection.oclIsUndefined())]
8     [generateEquationSection(inModel.equationSection)/]
9    [/if]
10   end [inModel.name/];
11  [/file]
12 [/template]
14
15 [template generateComponent(inC:Component) ? (inC.oclIsTypeOf(Part))]
16  ...
17 [/template]
```

Figure 6.4: Acceleo Template Example

The file keyword of the template means that a file is created each time the template is executed. Note that it is possible to add if and for statements to lead the code generation. Moreover, Acceleo offers the possibility to implement different behaviors for a template using preconditions. In Fig. 6.4, the question mark ? (line 15) defines a precondition: the template is executed only if the precondition is satisfied. To perform Modelica code generation, 29 Acceleo templates have been implemented.

6.2/ From SysML Models to BZP

In this section, we discuss the transformation of SysML models into BZP constraints. As depicted in Fig. 6.5, the overall transformation is divided into 3 stages. First, the SysML model is translated to a pivot model dubbed UML4TST. We decided to implement a pivot meta-model since BZP would not be the only targeted language for constraint reasoning. Indeed, according to the work of [Cantenot et al., 2012, Cantenot et al., 2013], it is possible to generate test cases using several SMT solvers (Z3, CVC3, CVC4, and MiniZinc) in a multi-threading fashion.

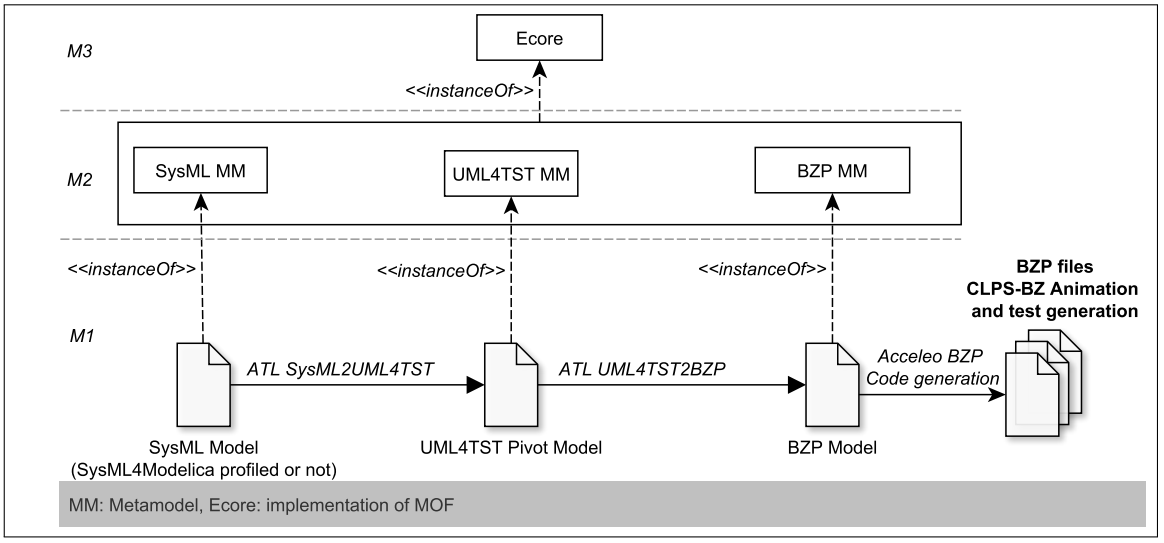


Figure 6.5: BZP Generation Process from SysML Models

The second step of out approach consists in translating the pivot UML4TST model to the BZP model. During this transformation, we also translate the OCL code into BZP. Finally, BZP files are generated with Acceleo. These three step are detailed in the next sections.

6.2.1/ SysML model to UML4TST model

The meta-model UML4TST have to satisfy the subset for animation described in the Chap. 5. This meta-model is depicted in Fig. B.2 and B.3. It contains all the information that concern classes, instances, state-machines, associations, links, and behaviours. Table 6.1 summarizes the mapping between SysML elements and out pivot meta-model. In order to transform SysML models into UML4TST models, we have implemented 13 helpers, 11 lazy rules, and 27 matched rules.

In addition, the OCL code contained in the model (guard, effect, pre / postcondition, bodyCondition, onEntry, and onExit) is parsed with ANTLR. Then, each evaluator of the parser instantiates the OCL meta-model of the Fig. B.4. This meta-model has been designed inside the UML4TST package. Indeed, we have created references between elements of the UML4TST meta-model and the OCL meta-model. In this way, we know which element of the UML4TST model is used in the OCL code.

The next step of our approach consists in translating the generated UML4TST-OCL model into BZP model.

6.2.2/ UML4TST model to BZP code

The BZP meta-model depicted in Fig. 6.6 enables to represent facts. These facts are then analyzed by the CLPS-BZ solver, which builds the graph of reachable states of the system described by the constraints. Note that each fact presented in Sect. 3.3.2 is represented as a meta-class. In order to transform UML4TST models into BZP models, we have implemented 16 helpers and 17 matched rules.

Table 6.1: Mapping Between SysML and UML4TST

SysML constructs	UML4TST constructs
Model	Model - Project
Package	Package
Block β: $\nexists\, \rho \in \Gamma_{Part} \mid \rho.type = \beta$	Class - Instance - Link
Block β: $\exists\, \rho \in \Gamma_{Part} \mid \rho.type = \beta$	Class
Enumeration	Enumeration
Enumeration Literal	Enumeration Literal
Property $\alpha.t \in \{\mathbb{Z}, \mathbb{B}, \Gamma_{Enum}\}$	Attribute
Operation	Operation
Operation.precondition	OCLBehaviour
Operation.postcondition	OCLBehaviour
Operation.bodyCondition	ActionOCLBehaviour
Parameter	Parameter
Association	Association - Link
Part	Instance
Region	StateChart
Pseudostate initial	InitialState
Composite state	CompositeState
State with submachine	CompositeState
State	SimpleState
ChoiceState	ChoiceState
FinalState	FinalState
State.onEntry (onExit)	ActionOCLBehaviour
Transition (local or external)	ExternalTransition
Internal Transition	Internal Transition
Transition.guard	OCLBehaviour
Transition.effect	ActionOCLBehaviour
CallEvent	Event

The OCL model is also translated into BZP using 22 helpers that return concatenated strings. Finally, the BZP model is the input of an Acceleo code generator. We have implemented 8 templates that generate BZP files from BZP models.

In the next section we present the implementation of the overall approach within an Eclipse framework.

6.3/ IMPLEMENTATION IN ECLIPSE

This section discusses the implementation of the overall process. Figure 6.7 shows the architecture of our simulation, animation, and testing environment from SysML models. This Eclipse-based tooled process is depicted in Fig. 6.8. It instantiates the intended process given in Fig. 3.1, and it strongly relies on Model-Driven Architecture (MDA) approach since model transformation and code generation procedures enable to automatically derive the simulation and testing artifacts from the SysML models [Gauthier et al., 2015a].

The first step of this approach (①) consists to model the SUT and the plant using the combining subset for simulation and animation. Papyrus[4] is used to support the SysML modelling. Then, the SysML model may be translated into Modelica and BZP constraints from the pop-up menu ②. The Modelica code can be modified from the view ③ as well as the generated BZP constraints from the view ④.

[4]https://www.eclipse.org/papyrus/

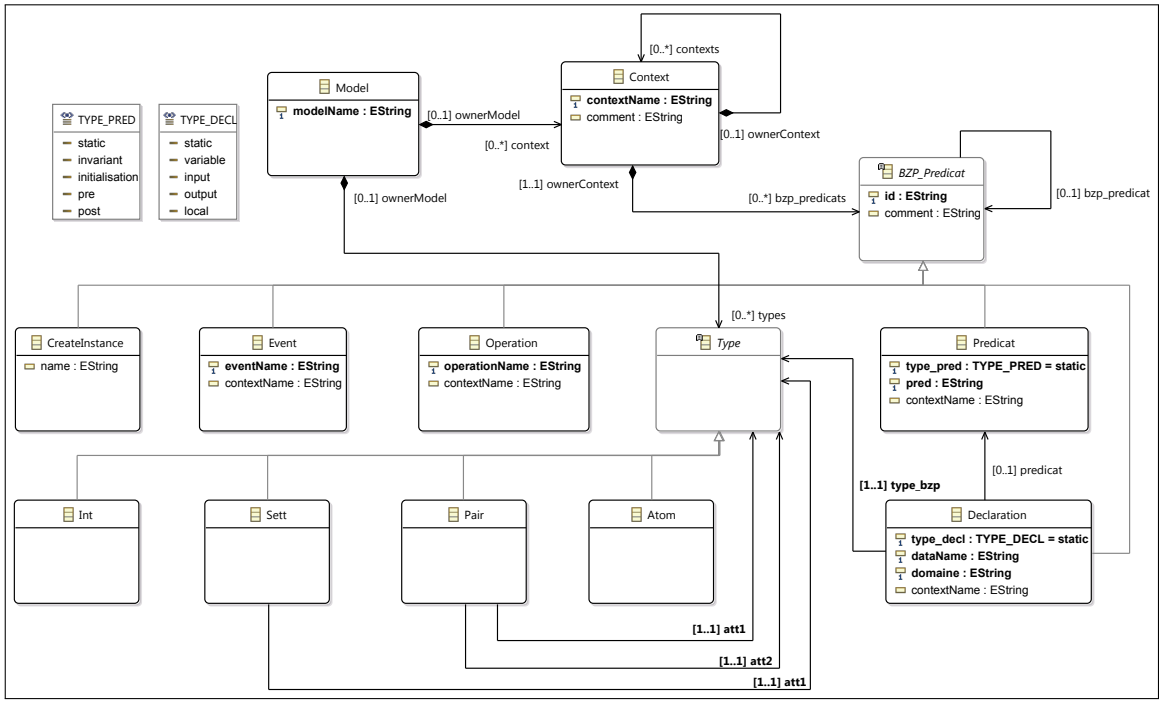


Figure 6.6: The BZP Meta-model

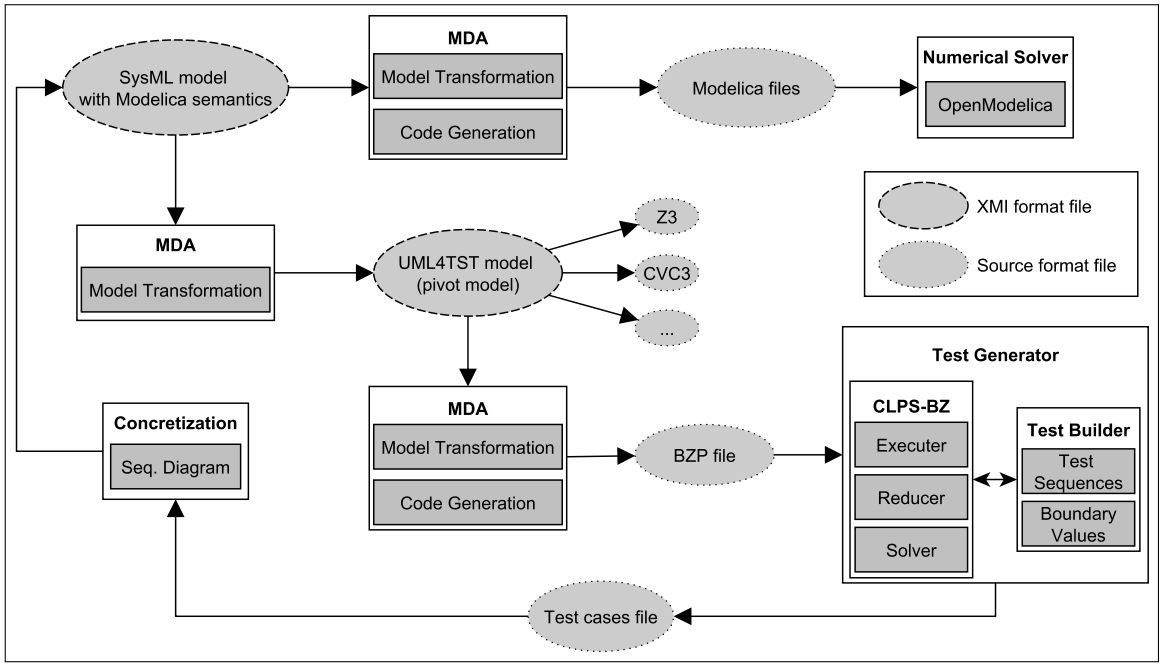


Figure 6.7: Overall Architecture of the Simulation and Testing Tooled Process

Finally, OpenModelica[5] computes the simulations, and CLPS-BZ (included as a plugin in our Eclipse environment) generates the test cases. Test cases are manually translated as sequence diagrams in the SysML model.

[5]https://www.openmodelica.org

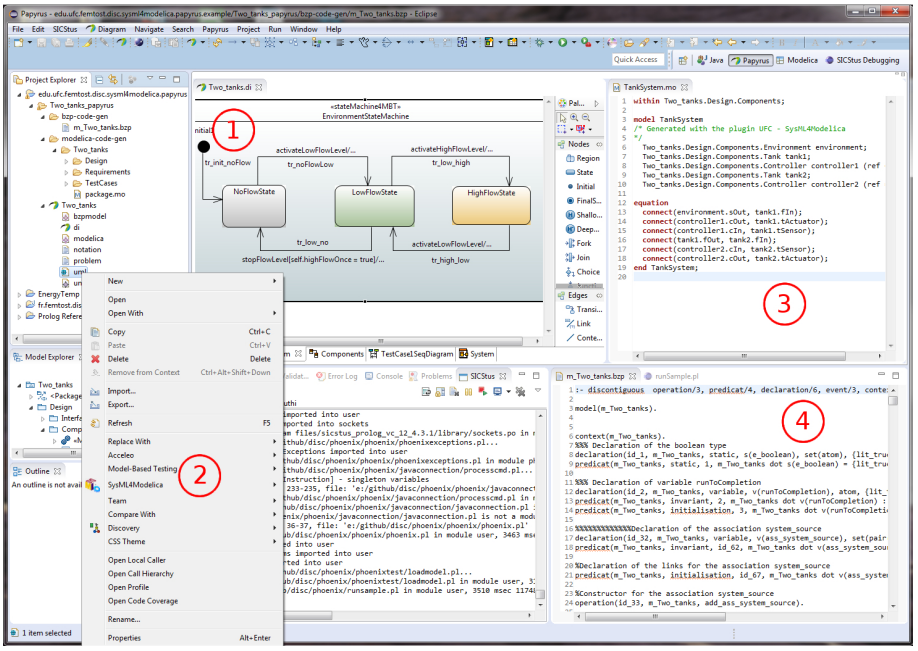


Figure 6.8: Implementation of the Proposed Process in Eclipse

The implementation of the OMG SysML-Modelica Transformation specification is available for the Papyrus environments. It is also adopted and promoted by the OMG SysML-Modelica working group. The whole process has been implemented over the Eclipse framework as a plugin for Papyrus. For further information, such as demos, examples and source code, we kindly refer the interested reader to the OMG SysML-Modelica Github[6].

6.4/ SYNTHESIS

This chapter has presented an approach to validate requirements of complex systems at the earliest stages of design process. To address this, we have implemented the OMG SysML-Modelica Transformation specification as a UML profile for SysML called `SysML4Modelica`. Therefore, we have proposed a new MDE process which enables to transform SysML models to Modelica models with ATL and to generate Modelica code from the Modelica models with Acceleo. Accordingly, we designed a novel Modelica meta-model that verifies Modelica syntax. Moreover, the constraints defined in the OMG specification are verified and thus ensure the SysML model consistency. Furthermore, the proposed implementation of the OMG SysML-Modelica Transformation specification is available for the Papyrus environments. It is also adopted and promoted by the OMG SysML-Modelica working group.

We have also investigated the combined use of model-based testing and simulation to validate hybrid systems. More specifically, we have implemented a SysML modeling approach that enables both to generate test cases and to simulate the system under test from the same SysML model. The test cases are generated from a BZP model, which is the resulting file of a translation between SysML and BZP constraints. This innovative approach has been validated with experiments on two case studies. These experiments are presented in the two next chapters.

[6]https://github.com/SysMLModelicaIntegration/edu.ufc.femtost.disc.sysml2modelica

7

SMARTBLOCKS CASE STUDY

Contents

In this chapter, we present the experiments and the obtained results about the modelling, code generation and simulation of a physical system named SmartBlocks. In the former project (Smart Surface), SysML language was used to model the system and VHDL-AMS was used to simulate it [Bouquet et al., 2012]. VHDL-AMS code was generated by MDE techniques but important instructions have not obvious matching with SysML constructs. Therefore, we decided to focus on the combined use of SysML and Modelica to perform simulation of complex systems.

This chapter is divided into four sections. First, we present the SmartBlocks system in Sect. 7.1. Then, we present the experimental motivations in Sect. 7.2. Section 7.3 introduces the mathematical model and the SysML model of this system. Finally, we present and discuss the simulation results in Sect. 7.4.

7.1/ SPECIFICATION SUMMARY OF THE SMARTBLOCKS

In this section we first introduce the SmartBlocks system. Then, we describe the experimental protocol to evaluate the issues raised by this case study.

The SmartBlocks case study is about a contact-free conveyor system that solves issues for the transport and the sorting of small and fragile objects. Indeed, this kind of object can be damaged by manipulations whereas clean products can be contaminated by the contact with the conveyor (especially in the pharmaceutical industries, microelectronic and food). To solve these problems, the robotics researchers of the FEMTO-ST[1] Institute are developing a new self-reconfigurable modular conveyor based on a contact-free technology (air-jet technology). This project is in the frame of the Project ANR-2011-BS03-005, named Smart Blocks.

[1] http://www.femto-st.fr/en/Research-departments/AS2M/Presentation/

This conveyor, illustrated in Fig. 7.1, is composed of 2.5 centimeters-size blocks which are linked together to form the conveying surface. Each block includes a MEMS (Micro Electro Mechanical Systems) actuator matrix (18 x 8) in the upper face in order to move the objects, sensors able to detect the object's position, a micro-controller, and some communication ports which link it with its neighbours in order to plan global transport policies. The conducted experiments focused on the modelling and simulation of the actuator array, which is composed of air-jet nozzles, and on its influence on a millimeter size object. The actuator matrix contains two kind of air-jets: conveying air-jets (represented as left to right arrows), and centering air-jets (represented as up-bottom or bottom-up arrows). For the need of the study we only considered the conveying air-jets.

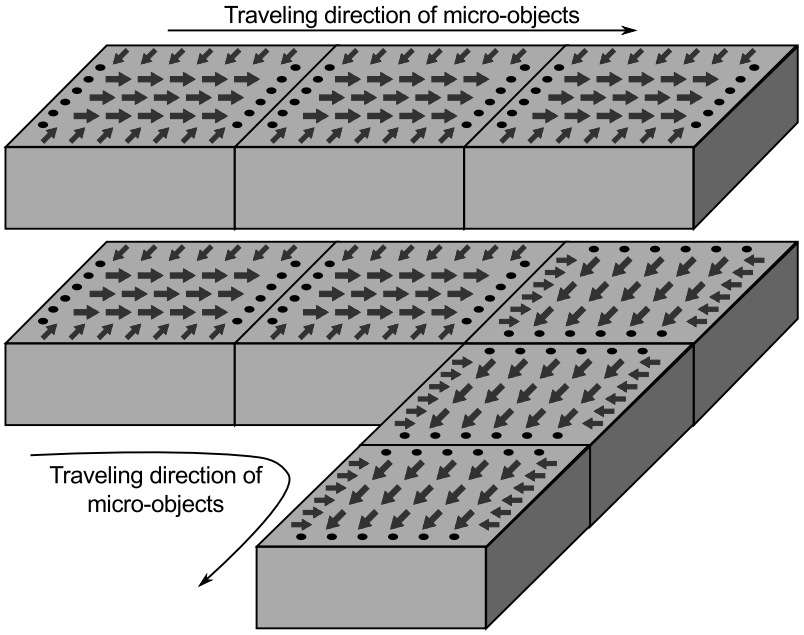


Figure 7.1: The SmartBlocks System

The main functional requirement of the smart block system is to convey small object with air-jets technology. Therefore, we need to predict and to master the behavior of a small object subjected to a high speed air flow. Modelling, simulation of air-jets are important tasks to understand how one small object can move from one point to another. Moreover, simulation could give us some clues on how to arrange the blocks to provide non-linear trajectories.

The domain experts gave us the requirements of Tab. 7.1 that mainly focus on the physical properties of the object during its displacement. For robotics engineers, these requirements raise a major issue that is to determine, according to different object configurations, if the air-jets force is too weak, too strong or sufficient enough.

7.2/ EXPERIMENTAL MOTIVATIONS

In this section we present the goal of the experimentation. To address this goal, we detail the protocol we used during the experimentation. Finally, we discuss potential threat to validity associated to the protocol.

Table 7.1: SmartBlock Requirements

ID	Description
Req 1	The system shall transport objects whose $1.e^{-6}kg \leq$ mass $\leq 5.e^{-3}kg$
Req 2	The system shall transport objects whose $1.e^{-4}m \leq$ diameter $\leq 1.e^{-2}m$
Req 3	The cruise speed of an object shall not exceed $6m.s^{-1}$
Req 4	The cruise speed of an object shall not be lower than $1m.s^{-1}$
Req 5	The acceleration of an object shall not exceed $5.5m.s^{-2}$

7.2.1/ GOAL OF THE CASE STUDY

For this case study, we focused on the modelling and the simulation of an object subjected to air-jets of one SmartBlock. Experiments on the SmartBlocks system were made in collaboration with a robotics engineer with no SysML and Modelica knowledge. Therefore, we used our prototype for carrying out the SmartBlocks case study in order to assess the relevance of the integration of continuous aspects in a SysML model and to answer the following questions:

1. In what extent a SysML model may be designed from a mathematical specification?
   - What are the required knowledge to perform continuous simulation from SysML models?
   - In what extent the proposed approach raise the communication between a system architect and a robotics engineer?
   - Is the specification clear enough to perform system decomposition?
2. How appropriate and convenient is the SysML4Modelica profile?
   - Does it add complexity during the modelling stage?
   - Does it permit to save a substantial amount of time?
3. Does it fit the need to validate a high-level SysML design and requirements at the soonest?

7.2.2/ EXPERIMENTAL PROTOCOL

We have divided this experiment into four stages:

1. analysis of the mathematical specification to extract sub-systems,
2. SysML model proposal of the system,
3. simulation of a SmartBlock: constant initial conditions over the object (constant mass and diameter), involved line of 8 air-jets as parameter (1 line, 3 lines, and 5 lines),
4. simulation of a complete SmartBlock (8 lines of 18 air-jets) whilst playing on the object's parameters and feedback from the domain experts.

7.2.3/ THREAT TO VALIDITY

First, it should be noted that these experiments have been conducted by an engineer who is familiar with SysML modelling but without any initial knowledge about real-time simulation. In addition, the engineer is the one who has developed the prototype (see Sect. 6). This means that we have to put things into perspective as well as to consider a system architect who did not developed the proposed approach. In this way, it might bias us to have an objective view of the applicability of the process. Therefore, to assess the relevance of the proposed approach, we have to consider an engineer with no background about Modelica. However, case-study results have been evaluated with scientists specialized in mechatronics systems, who are therefore familiar with development and continuous simulation of such complex systems. This enables us to get a solid feedback regarding the relevance of the framework and the related tool-supported overall approach.

The second threat to validity is linked to the first one. The gap between SysML and Modelica being quite important at business-level, the cost to perform simulation from SysML model can be also important since design teams may have to learn two languages. This could be somehow an issue for existing processes in the industry. Engineers that are familiar with the use of SysML should learn the basics of Modelica to correctly apply the SysML4Modelica profile and to write equation with a good syntax.

The third identified threat to validity concerns the complexity and the criticality of the SmartBlocks system. Since the perimeter of the study was limited to the air-jet technology, we did not designed a model containing multi-physical components: electrical components for instance, such as sensors or controllers. Strickly speaking, the system under study is neither complex or critical. However, this study will give us some feedback regarding the applicability of the approach for more complex and critical systems.

7.3/ MATHEMATICAL AND SYSML MODELLING

This section presents the mathematical model of each components. The mathematical equations and the specification were given by the domain expert. The resulting SysML models is then presented.

7.3.1/ THE OBJECT

To model the behaviour of the object subjected to the propulsion of several air-jets, we consider the elementary force of one air-jet in two dimensions. Indeed, we consider the following hypotheses:

- levitation is provided by the air jets below the object but we don't consider their effect, i.e. they are not evaluated,
- air-jets are independent, i.e. there aren't any interactions between air-jets.

Then, we make the sum of each air-jets with their degree of influence. That influence, depends on the position of the air-jet in regard to the position and the distance with the object. Overall, the object is subjected to two forces: a driving force $\overrightarrow{F_d}$ and an opposition force of displacement $\overrightarrow{F_v}$ (air friction). Their scope is represented in Fig. 7.2.

$$\sum \overrightarrow{F} = \overrightarrow{F_d} + \overrightarrow{F_v} = m \cdot \ddot{x} \cdot \overrightarrow{u_n} \tag{1}$$

m: weight of the object, x: position of the object.

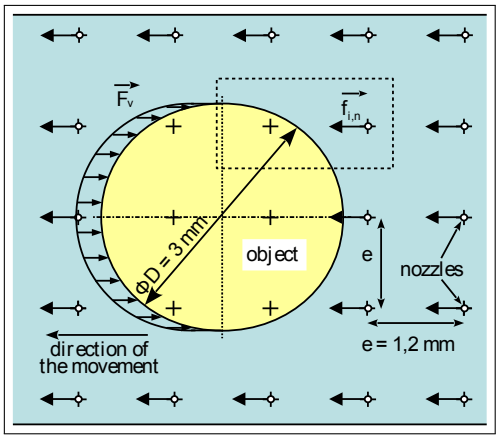


Figure 7.2: Object Subjected to Air-jets

The driving force $\overrightarrow{F_d}$ results to the sum of air-jet propulsion forces which act near to the object.

$$\overrightarrow{F_d} = \sum_{i=1}^{M} \sum_{n=1}^{N} \Delta_{i,n} \cdot \overrightarrow{f_{i,n}} \tag{2}$$

$\Delta_{i,n} = 1$ if the air-jet reach the surface of the object.

In opposition to the displacement appear viscous drag forces $\overrightarrow{F_v} = \int_S \overrightarrow{f_v}$ represented as:

$$\overrightarrow{F_v} = -K \cdot \eta \cdot \dot{x} \cdot \overrightarrow{u_n} \tag{7}$$

$K = 2,75mm$: geometric coefficient of the viscosity force
$\eta = 1,81.10^{-}5$: air viscosity

Figure 7.4 shows the resulting SysML block of the object. This block contains a flow port that models the received driving force from SmartBlocks. In addition, the flow port `object_position` enables to give the object's position to the system.

The Equations (1) and (2) are specified as depicted in Fig. 7.3. Note that the flow port `airjet_forces` is an array of connector. Indeed, it enables to get the resulting air-jet force of each SmartBlock connected to the object.

```
sum(tab_forces) + air_friction.val = der(x_velocity) * mass; (constraint 1)

air_friction.val = -k * nu * der(x); (constraint 2)

der(x) = x_velocity; (constraint 3)

object_position.xval = x; (constraint 4)

object_position.yval = y; (constraint 5)

for i in 1:nbBlock loop
 tab_forces[i] = airjet_forces[i].val; (constraint 6)
end for;
```

Figure 7.3: SysML Constraints for the Object (Modelica Subset)

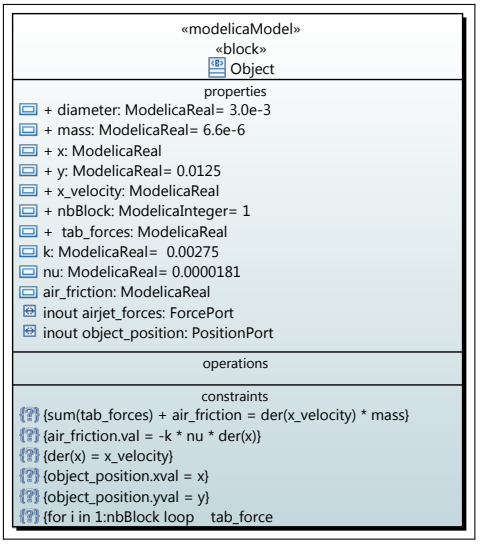


Figure 7.4: SysML Block of the Object

7.3.2/ THE SMARTBLOCK

As stated in Sect. 7.1, a SmartBlock is a 2.5 centimeters-size block, which includes an actuator matrix, sensors, a micro-controller, and some communication ports. The experiment focused on the modelling and simulation of air-jets. Hence, we only considered the actuator matrix. However, this matrix had to be positioned precisely over the SmartBlock. As depicted in Fig. 7.5, we considered the position of the sensors and the position the centering air-jets to model a realistic conveying surface.

7.3.3/ THE ACTUATOR MATRIX

The actuator matrix is composed of 18 x 8 air-jets length. The direction of air-jets is defined through unx and uny variables. The positioning of the matrix on a SmartBlock is defined with coordinates x, y. Figure 7.6 shows the dynamic positioning of each air-jet over the matrix (constraint 1). In addition, each air-jet's influence is saved in an array tab_forces. Hence, the global force is the sum of each elementary air-jet's force (constraint 2). Note that, if $unx = 1$ then the object moves forward and if $unx = -1$ then the object is slowed or moves backward. The resulting SysML block is depicted in Fig 7.7.

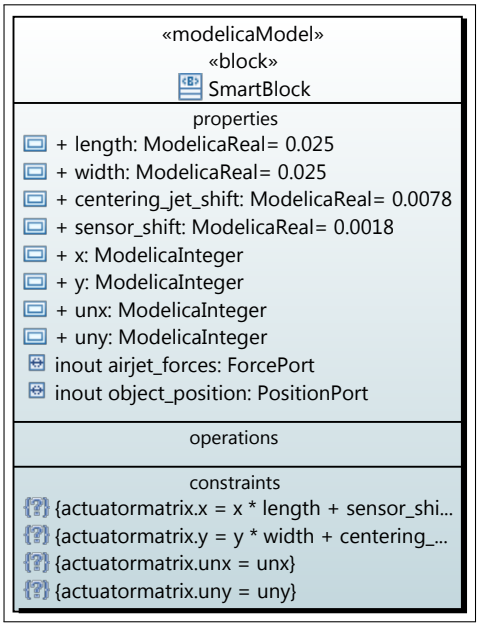


Figure 7.5: SysML Block of a SmartBlock

```
for i in 1:nb_airjet_length loop   (constraint 1)
  for j in 1:nb_airjet_width loop
    airjets[i,j].x = x + i * 0.0012;
    airjets[i,j].y = y + j * 0.0012;
    airjets[i,j].unx = unx;
    airjets[i,j].uny = uny;
  tab_forces[i,j] = airjet_forces[i,j].val;
  end for;
end for;

applied_forces.val = sum(tab_forces) * unx; (constraint 2)
```

Figure 7.6: SysML Constraints for the Actuator Matrix (Modelica Subset)

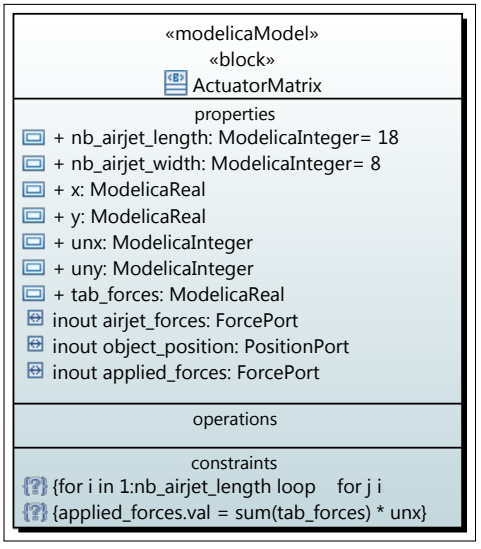


Figure 7.7: SysML Block of the Actuator Matrix

7.3.4/ THE AIR-JET

The elementary force $\overrightarrow{f_{i,n}}$ (Fig. 7.8) induced from each air-jet is determined as follows [Matignon et al., 2010]:

$$\overrightarrow{f_{i,n}} = \frac{1}{2}\rho \cdot C_D \cdot s_{i,n} \cdot v_{i,n}^2 \cdot \overrightarrow{u_n} \tag{3}$$

$\rho = 1,3kg/m^3$ if the air-jet reach the object surface
$C_D = 1,2$ drag coefficient for an half-cylinder
$s_{i,n}$: projected surface in contact of the air-jet
$v_{i,n}$: relative speed of the air-jet, defined as:

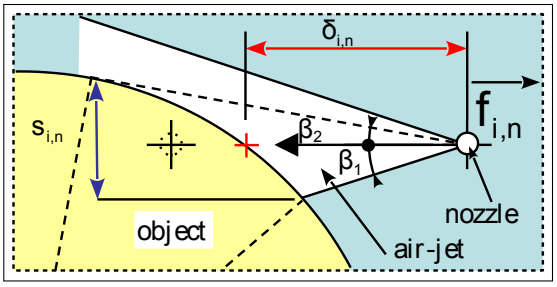


Figure 7.8: Model of one Air-jet

A major issue was raised during the specification of the projected surface $s_{i,n}$. Indeed, this value is the key stone of the under approximation of the problem since we approximate the arc of circle subjected to the air-jet by the projected surface $s_{i,n}$.

To determine the value of $s_{i,n}$ at each time step, we have applied the algorithm 1, considering that the air-jet has an angle of $\frac{\pi}{6}rad$. Because s is a line, we can characterized it with two points: $A = (s.x_{sup}, s.y_{sup}), B = (s.x_{inf}, s.y_{inf})$. Therefore, the algorithm first computes the intersection between the air-jet upper and lower limits with the object. If there is no intersections, then the algorithm computes the tangents (doted line from the air-jet in Fig. 7.8) and its intersection with the object. The two intersections, finally give us the two points A and B that are used to calculate the value (length) of s.

The dynamic combination of these functions is not easy because the object is always in movement and the speed is not continuous. The trajectory control depends on the position of the object onto each blocks. We have defined seven SysML `FunctionBehaviors` to calculate the projected surface s. These functions own a body attribute in which we have written Modelica code to perform the calculation of the Algorithm 1.

The relative velocity of the air-jet depends on the object's velocity and on the air-jet speed at the nozzle outlet:

$$v_{i,n} = \dot{x} - v_{air}(\delta_{i,n}) \tag{4}$$

v_{air} is the absolute speed of the air-jet and can take two values:

$$\delta_{i,n} > 0 \Rightarrow v_{air}(\delta_{i,n}) = 5500 \cdot \exp^{-\frac{\delta_{i,n}^2}{4}} \tag{5}$$

$$\delta_{i,n} = 0 \Rightarrow v_{air}(\delta_{i,n}) = 0 \tag{6}$$

$\delta_{i,n}$: distance between the air nozzle and the contact point of the object.

Algorithm 1 Calculation of the Projected Surface

Input:

$o.(x, y)$, position of the object
$j.(x, y)$, position of the air-jet
r, radius of the object

Output:

s , the projected surface

procedure PROJECTEDSURFACECALCULATION
  $\Delta_{sup} \leftarrow getIntersection$(upper limit of the air-jet, object)
  $\Delta_{inf} \leftarrow getIntersection$(lower limit of the air-jet, object)
  if $\Delta_{sup} < 0$ **then** ▷ The upper intersection does not exist
    $s.x_{sup} \leftarrow getIntersection(tangentSup, object)$
  else ▷ The upper intersection exists
    $a \leftarrow \tan\frac{\pi}{6} * \tan\frac{\pi}{6} + 1$
    $b \leftarrow 2 * \tan\frac{\pi}{6} * j.y - 2 * \tan\frac{\pi}{6} * \tan\frac{\pi}{6} * j.x - 2 * \tan\frac{\pi}{6} * o.y - 2 * o.x$
    $s.x_{sup} \leftarrow getRoot(\Delta_{sup}, a, b)$
  if $\Delta_{inf} < 0$ **then** ▷ The lower intersection does not exist
    $s.x_{inf} \leftarrow getIntersection(tangentInf, object)$
  else ▷ The lower intersection exists
    $a \leftarrow \tan\frac{\pi}{6} * \tan\frac{\pi}{6} + 1$
    $b \leftarrow 2 * \tan\frac{\pi}{6} * o.y - 2 * \tan\frac{\pi}{6} * j.y - 2 * \tan\frac{\pi}{6} * \tan\frac{\pi}{6} * j.x - 2 * o.x$
    $s.x_{inf} \leftarrow getRoot(\Delta_{inf}, a, b)$
  $s.y_{sup} \leftarrow \tan\frac{\pi}{6} * s.x_{sup} - j.x * \tan\frac{\pi}{6} + j.y$
  $s.y_{inf} \leftarrow -\tan\frac{\pi}{6} * s.x_{inf} + j.x * \tan\frac{\pi}{6} + j.y$
  $s \leftarrow abs(s.y_{sup} - s.y_{inf}$
return s

Figure 7.9 shows the resulting SysML block of the air-jet. This block contains a flow port that receives the object's position. The object's position permits to calculate the elementary force $\overrightarrow{f_{i,n}}$ of each air-jet.

The whole BDD of the system is depicted in the Fig B.5 (Appendix B). Note that the `ActuatorMatrix` only contains one part named `airjets`. However, a matrix is composed of 144 air-jets. In addition, we have to consider a system with several SmartBlocks. We did not created a set of 144 parts inside a the actuator matrix. Indeed, we have used the property `arraySize` provided by the `<<modelicaPart>>` stereotype. This property allows us to define a two-dimensional array bounded by the `nb_airjet_length` and `nb_airjet_width` attributes of the block `ActuatorMatrix`. Hence, we are able to detect if the `arraySize` property is set, and to generate Modelica array of air-jets. Each air-jet is then connected with a `for` statement.

Figure B.6 shows the internal view of the system. This view permits to represent the interactions between components which compose the system, to say forces and position of the object. Because the object is subjected to the force $\overrightarrow{F_v}$, it's block is linked with the block *SmartBlock*. In addition, each conveyor block has to know the position of the object in order to calculate the force of the air-jets.

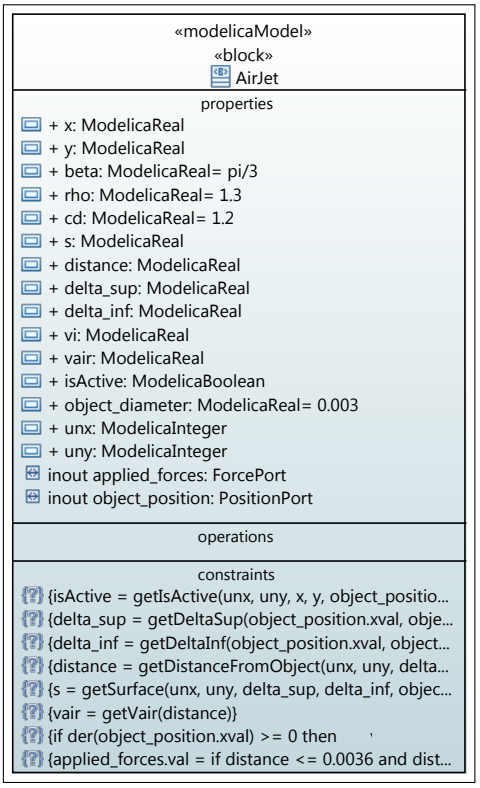


Figure 7.9: SysML Block of an Air-jet

Figure B.7 depicts the blocks that serve for flow ports typing. In this system we distinguish 4 kind of flowing information that need to go through components: real values with the *RealPort*, a Newtonian force with *ForcePort*, coordinates of the object with *PositionPort*, and Boolean value *BooleanPort*. Note that the real port and the Boolean port are not used in this IBD. They serve to connect the sensor and the controller.

This section has described the model of the Smart Block at a mathematical level and system level. The last step of our work is to simulate the model in order to validate it and to study the influence of the air-jets on the object. We have developed a plugin for Papyrus which can automatically translate the SysML model into Modelica code. Therefore, we present simulation results directly from the execution of the model in the OpenModelica environment.

7.4/ RESULTS OF THE EXPERIMENTATION

As a reminder, a block is composed of a conveying surface (named *ActuatorMatrix* in the SysML model). This surface is a matrix of 18 air-jets long and 8 air-jets width. The first phase of this experiment is the simulation of three scenarios to understand the air-jets influence on the object. First, the object is subjected to one line of air-jets, then to three lines and finally to five lines of air-jets. For each simulations, the object is centered on the surface. These scenarios, which are illustrated in Fig. 7.10, allow to validate the model in regards of the expected behaviors of the object.

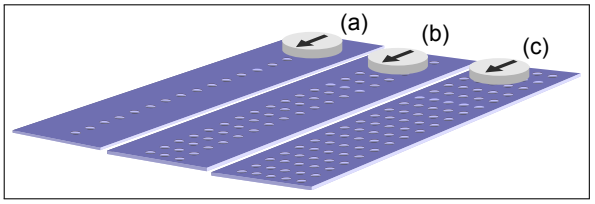


Figure 7.10: Simulation Scenarios

Then, we started our first experiment under the following initial conditions:

- mass of the object : 6.6e-6 kg
- size of the object: 0.003 m in diameter
- initial velocity of the object: 0.0 m/s

The goal of these simulations is to study how a very light object behave at the very beginning of the transfer process. Therefore, we have focused the study on a period of 5ms. As shown in Fig. 7.11, the simulation results focus on the position, the velocity and the acceleration of the object. While the object is moving, it successively losts and gains driving forces from the air-jets.

On simulation results 7.11 (a), only one or two air-jets are influential at a time. This results to a 100% variation of forces. That's why we can observe jerks on the acceleration curve. On simulation results 7.11 (b), a curve smoothing is observed. Indeed, there are successively 3 to 4 and 4 to 6 influential air-jets at a time, which implies a 50% variation of forces.

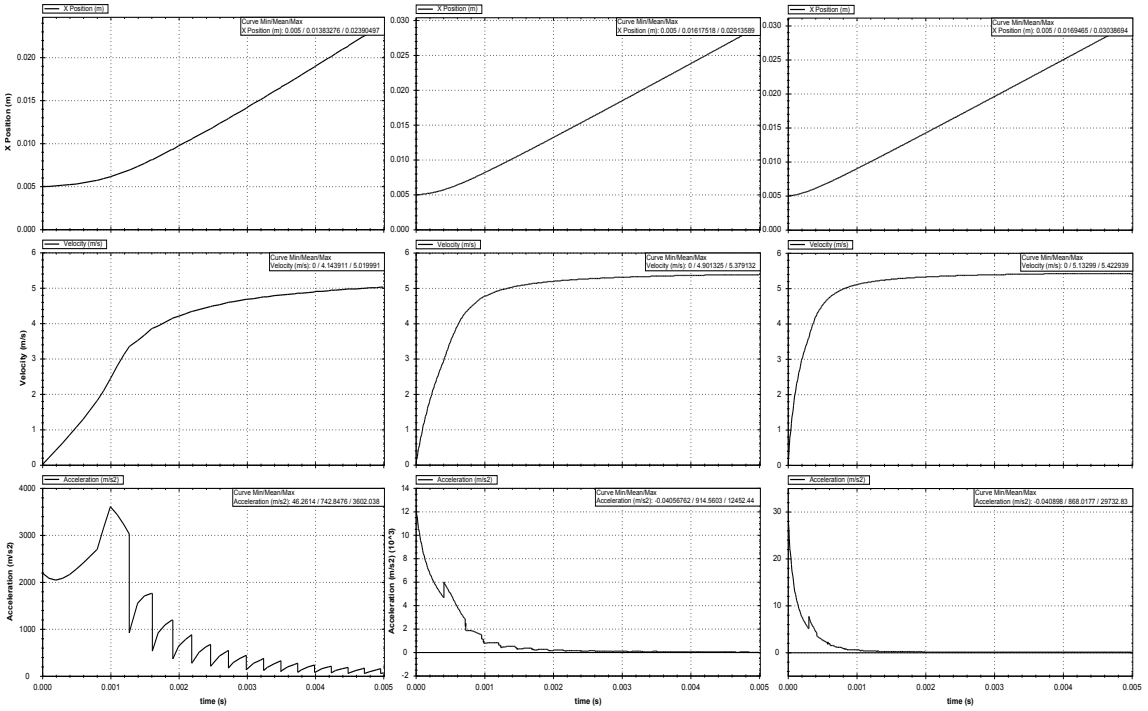
Concerning the last scenario, which results are illustrated in Fig. 7.11 (c), we can observe a more important smoothing effect on the acceleration curve. We have also simulated an object subjected to 8 lines of air-jets. But, since the model takes into account the distance between the object and the air-jets, only the nearest air-jets influence the x position of the object. Therefore, the simulation results are identical for five or more lines of air-jets.

The Table 7.2 summarizes the times to simulate 5ms of each scenario. For instance, simulating 5ms of a SmartBlock composed of 5 lines of 8 air-jets took 156 seconds (including output file creation, event handling, and overhead). The long simulation time is due to the huge number of equations involved during the numerical integration. For instance the last scenario gives rise to 1735 equations, that have to be resolved at each time step.

Table 7.2: Simulation Times of each Scenario

Scenario	Creating output file	Event-handling	Overhead	Simulation	Total
1 line	3.6459 s	0.34958 s	0.2394 s	12.6742 s	16.90908 s
3 lines	7.4895 s	1.3273 s	0.96784 s	47.98204 s	57.76668 s
5 lines	16.34091 s	4.20543 s	1.8532 s	134.3409 s	156.74044 s

We noticed that the simulation results did not satisfy the requirements Req 5 of the Table. 7.1. Indeed, the acceleration of a very light object (6.6e-6 kg) is higher than $500m \cdot s^{-2}$. Therefore, we have made a second experiment to determine the limits of the system regarding the size and the weight of the object to transport.



(a) One line of air-jets (b) Three lines of air-jets (c) Five lines of air-jets

Figure 7.11: Modelica Simulation Results

The second part of the experiment, was the Modelica code generation and simulations using the initial conditions given in Tab. 7.3. Therefore, the second part of the experiment consisted in modifying the parameters of the object in order to evaluate the model considering the requirements of the Tab. 7.1. For these simulations, we have considered a SmartBlock of 8 lines of air-jets (identical simulation time for each scenario).

Table 7.3: Initial Conditions for Simulation

Scenarios	Object Mass	Object Diameter
S1	m = 5.$e^{-3}kg$	d = 0.01m
S2	m = 1.$e^{-3}kg$	d = 0.01m
S3	m = 1.$e^{-3}kg$	d = 5.$e^{-3}m$
S4	m = 5.$e^{-4}kg$	d = 5.$e^{-3}m$
S5	m = 5.$e^{-4}kg$	d = 1.$e^{-3}m$

As depicted in Fig. 7.12, each scenario verifies the requirements over the cruise speed. However, we can see in Fig. 7.13 that the scenario S4 7.13 (c) does not meet the requirement Req 5: the acceleration is indeed higher than 500 m/s^2. Consequently, there is a need to control air-jets depending on the properties of the object to convey, what constitutes a crucial constraint to be verified before building physical Smart Block system device. The decision was to add a controller that limits the air-jets force. In this way, depending on the acceleration, the controller must be able to start and stop the air-jets very quickly.

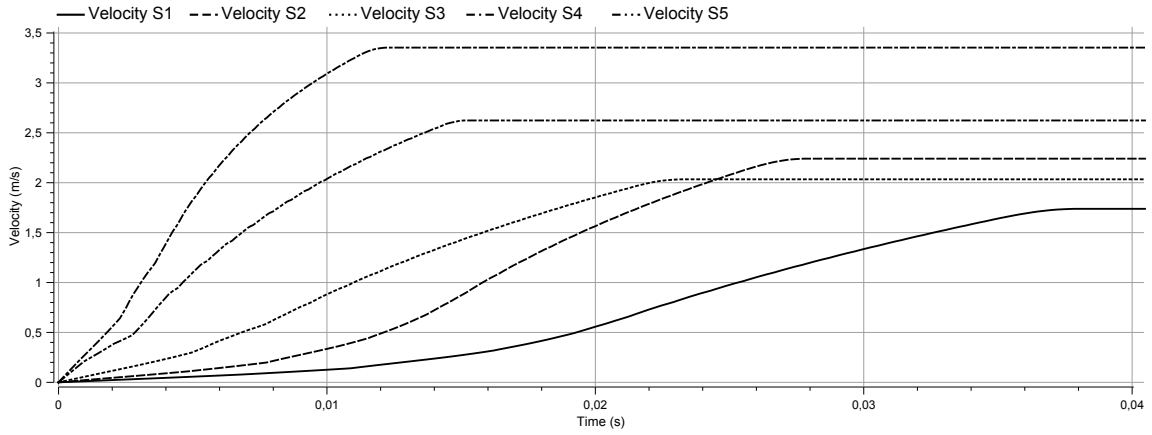


Figure 7.12: Object's Velocity for Each Scenario

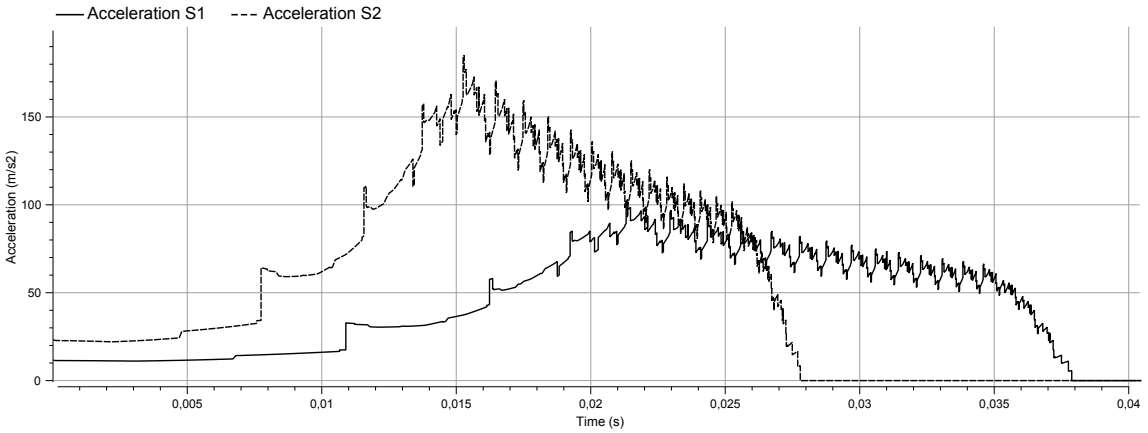
Despite strong starting assumptions, especially the absence of influence of an air-jet on the others, we consider that the model is a good starting point for other scenarios. For instance, we started to explore the braking capabilities of a smart block positioned in the opposite direction to the object's trajectory. We have observed that only few air-jets are sufficient to slow down the object efficiently.

7.5/ DISCUSSION

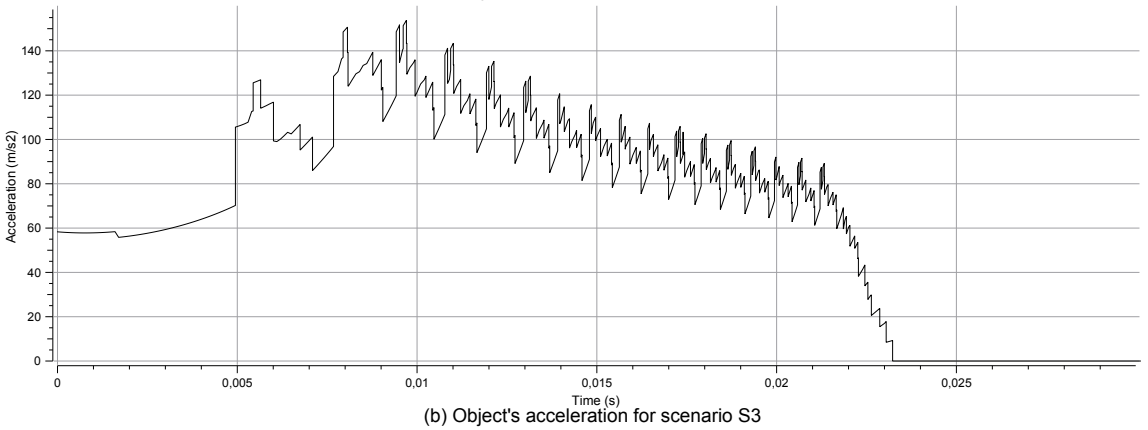
At the first sight, the relevance of the proposed approach to early validate system design using SysML models to perform Modelica simulation, instead of writing directly Modelica code for simulation, is debatable. To feed the debate, we will consider two key roles: a system architect and a robotics engineer. Consider now that the system architect early takes care of the validation process and has knowledge in computer science and model-based testing, while the robotics engineer has knowledge in numerical simulation and physics. Now the question is: what does the proposed approach bring to them?

From the robotics engineer point of view we may extract two benefits. First the proposed approach allows the robotics engineer to focus on the mathematical specification without taking care with the implementation. In the context of safety-critical systems, the previous assertion is true if the system engineer is able to provide generated and certified simulation code (certified C or ADA code may be generated from simulation models in the SCADE suite[2]). The second asset concerns communications between the system engineer and the robotics engineer. Considering that the first benefit is true, early feedback are possible between the two engineers. Indeed, since model calibration is a crucial stage, it becomes possible to concentrate the effort on the simulation results analysis and on design choices rather than on how to implement it. The SmartBlock case study illustrate those assertions. Indeed, this case study is managed by strong requirements concerning the velocity and the acceleration of different kind of tiny objects. Therefore, the SysML model of the system had to meet these requirements. We have performed several simulations with different initial conditions and the results gave our partner some clues on future design choices.

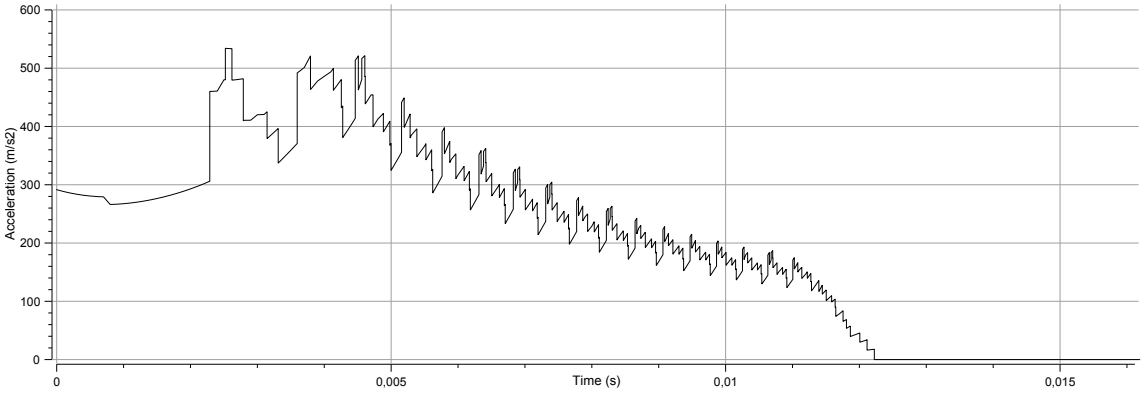
[2]http://www.esterel-technologies.com/products/scade-suite/automatic-code-generation/scade-suite-kcg-ada-code-generator/



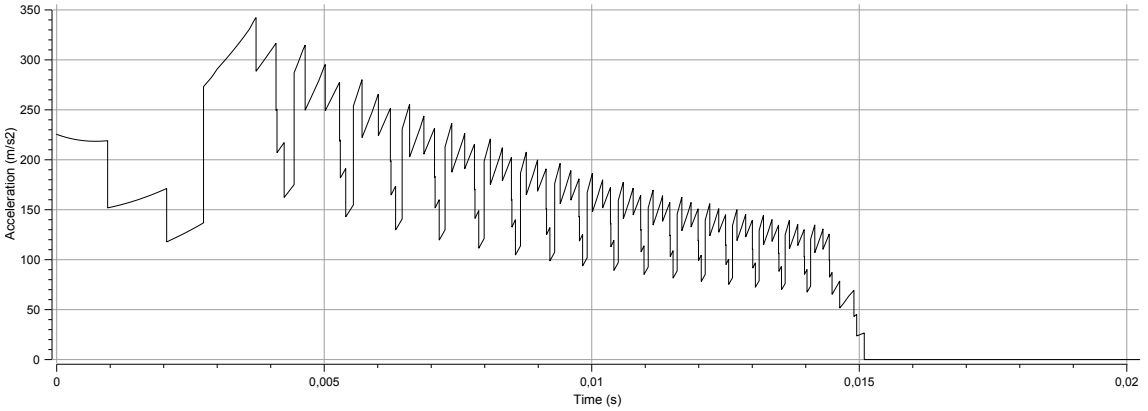
(a) Object's acceleration for scenario S1 and S2



(b) Object's acceleration for scenario S3



(c) Object's acceleration for scenario S4



(d) Object's acceleration for scenario S5

Figure 7.13: Object's Acceleration for Each Scenario

From a validation engineer point of view, the proposed approach seems to add complexity: learning the basics of the targeted simulation language and knowledge in mathematics and physics (such as integration and differentiation) are required. However, from this experiment, we may assess that the efforts spent to learn the basics of Modelica and to apply the SysML4Modelica profile were weak compared to the time saved by automatically generating the simulation code. Indeed, learning the subset for Modelica equation should be sufficient to perform early simulation from a SysML model. It should also be underlined that the SysML4Modelica profile can be applied to existing SysML models without changing the overall structure of the model. Incomplete model can also be handled: we could generate the structure of Modelica code without taking into account all the behavioral aspects (equations, algorithms, etc).

Finally, the case study presented in this chapter is not very large. To evaluate the scalability of the proposed approach, we have applied it on a large and complex electrical power system of a new generation of aircraft. This case study is the topic of the next chapter.

8

ELECTRICAL POWER SYSTEM WITH ENERGY MANAGEMENT

Contents

The tool chain presented in the Chapt. 6, has been tested out during the prototyping phase of an Electrical Power System (EPS) within HIL process. Hence, we distinguish the system under design (the EPS) and its environment called the plant system (a simulation model of some aircraft's instruments). The objective of our experiment was to assess the suitability and the reliability of the combined formalism to perform simulation and test generation.

This chapter is organized as follows. Sect. 8.1 introduces the informal specification of the system. Then we present the experimental motivations in Sect. 8.2 In Sect. 8.3, we detail mathematical model and the SysML model of the EPS system. Section 8.4 reports on the results obtained from this case study. Finally, we discuss the obtained results in Sect. 8.5.

8.1/ SPECIFICATION SUMMARY OF THE EPS

In this section, the specification of system is presented. First, the electrical architecture of the model is detailed. Then a representation thereof is proposed, which enables the structuring of its model and the identification of setting parameters. Finally, the control principles of the system and the energy management are presented.

8.1.1/ ARCHITECTURE OF THE SYSTEM

The electrical diagram of the system is as follows:

The overall system, depicted in Fig. 8.1, is composed of the following components:

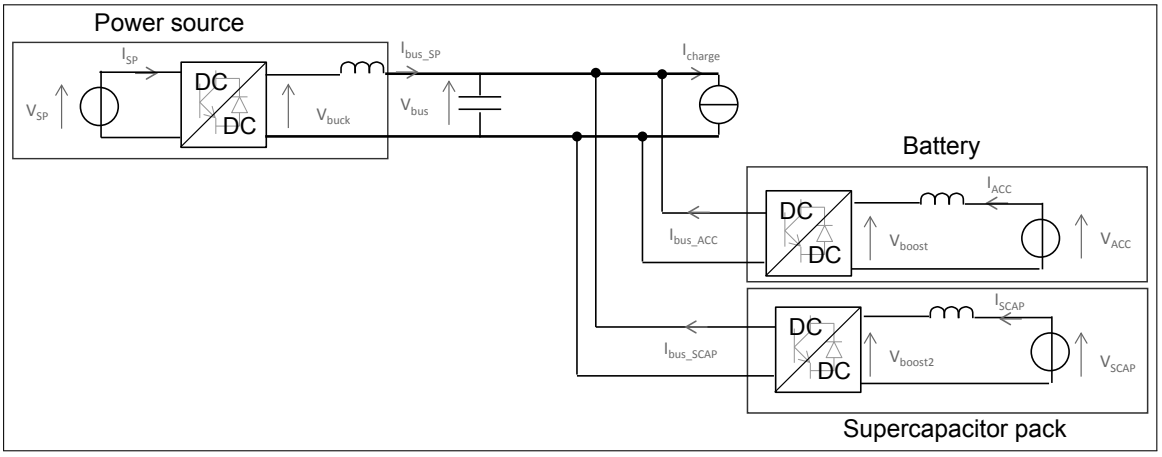


Figure 8.1: Electrical Diagram of the System

- a power source (SP) connected to the network via a DC/DC converter,
- electrical storage components:
  - a battery (BAT) and its DC/DC converter connected to the bus,
  - a super-capacitors pack (SCAP) and its DC/DC converter connected to the bus,
- a voltage bus,
- some consumers (electrical load).

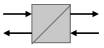
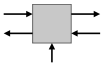
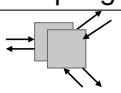
For this experiment we consider the system under test, or under validation, as being the system composed of the power source, the battery, the super-capacitor pack, and the bus. The plant concerns aircraft instruments, i.e. the electrical loads that need energy. In the rest of this chapter, we refer the Electrical Power System as the EPS.

8.1.2/ REPRESENTATION OF THE SYSTEM

The system, as described in the previous section, is a relatively complex system including components of different nature and field. To organize the modelling phase of this system, it exists various methodologies, including the Causal Ordering Graph (COG) [Hautier and Barre, 2004], the Bond Graph [Sueur and Dauphin-Tanguy, 1991] or Energetic Macroscopic Representation (EMR) [Bouscayrol et al., 2006]. The latter was chosen by the domain experts for the specification of the EPS because it is particularly suitable for the study of multi-physics and multi-scale systems. Indeed, this formalism provides a homogenous synthetic graphical representation of a complex system. Based on the principle of action/reaction and integral causality, this methodology provides a way of structuring a model and of identifying the setting parameters of a system. It also enables to perform system decomposition and thus to identify the sub-systems.

The main elements, particularly those that are used in this case study, are summarized in the Table 8.1. An energy source is a generator source or a receiver, imposing state variables on a system. An energy storage is a non-instantaneous conversion element,

Table 8.1: Energetic Macroscopic Representation Elements

Element	Energy source	Energy storage	Energy converter	Coupling
EMR				

assuming an inherent causality and considering the temporary energy storage function. An energy converter (with or without control variable) is an instantaneous conversion element providing a modulation of a power sample without admitting external variables and causality. A coupling element represents the distribution of power flow.

The energetic macroscopic representation of the EPS is given in Fig. 8.2. Sources and electrical loads are represented by an oval pictogram (electric sources, generating for the power source and receiving for the electrical loads). The accumulation elements (inductive or capacitive) are represented by crossed rectangles. Conversion elements are represented by rectangles. The bus is realized by an electrical coupling element (superposed rectangles) and a storage element, which is equivalent to a capacitor. All sources are then considered as energy sources from the bus point of view.

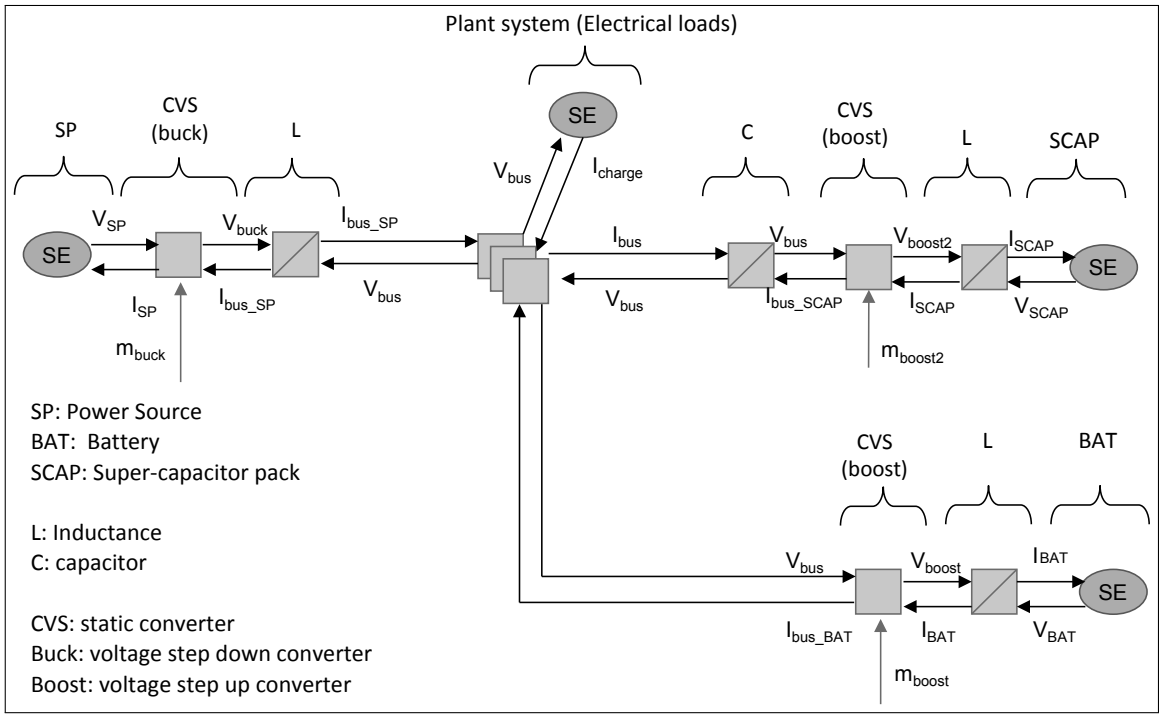


Figure 8.2: Energetic Macroscopic Representation (EMR) of the EPS

The EMR applied to the system shows the setting parameters to manage power flow (represented by the arrows mbuck, mboost, and mboost2). These variables enable to handle the distribution of energy between the different sources.

8.1.3/ CONTROL OF THE SYSTEM

One of the benefits of EMR methodology is the deduction of a control structure by systematic inversion model. Figure 8.3 illustrates a maximum control structure (MCS) [Locment and Sechilariu, 2010] to regulate the bus voltage. The control of the bus voltage is obtained by inversion of the capacitor. This regulation minimizes the difference between the measurement and the reference and defines a current reference for the SCAP (voltage and current control loops are interlinked). When an energy management strategy is implemented, reference currents of other sources are determined in a supervisor.

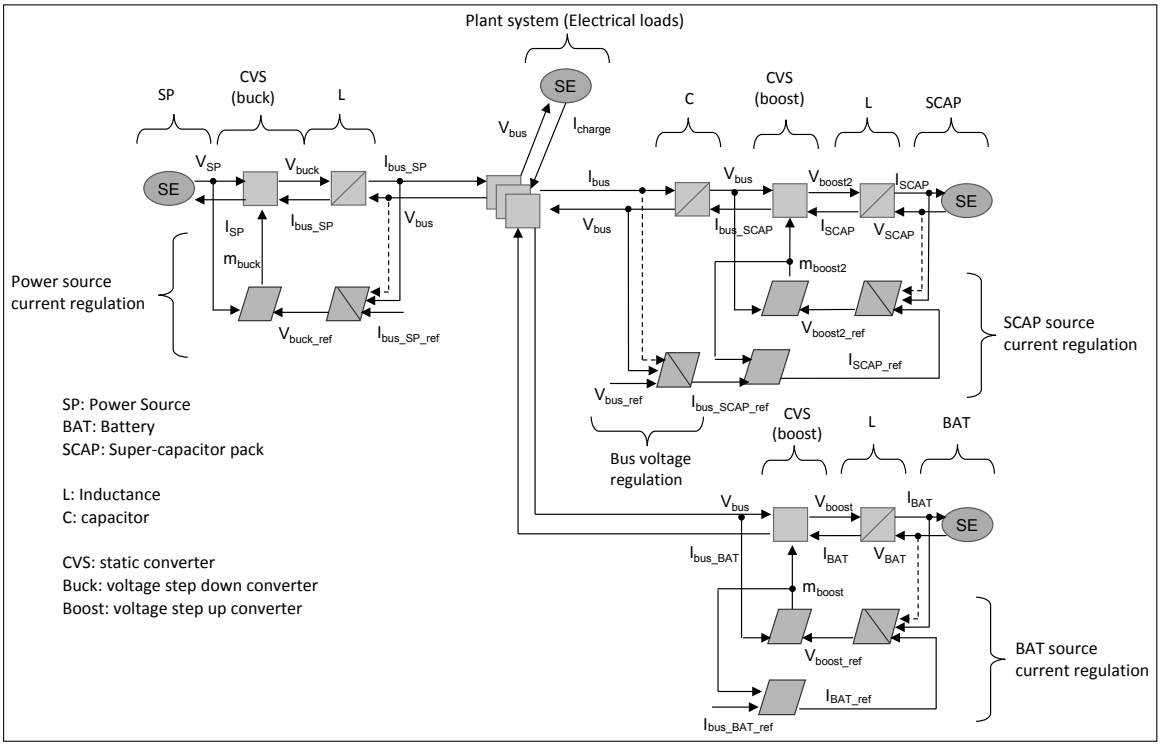


Figure 8.3: EMR and Maximum Control Structure of the EPS

Generally, the bus voltage regulator is located closer to the most dynamic source (the SCAP module), which enables a better efficiency of the voltage regulation. In principle, the bus voltage is regulated according to a reference V_{bus_ref} (typically 18V). Regulating the bus voltage is obtained as follows:

- by modifying the setting parameters of the converters associated to:
  - storage elements BAT and SCAP
  - the power source SP
- by balancing at every time step:
  - the currents supplied by the sources: I_{bus_SP}, I_{bus}, and I_{bus_SCAP}
  - the current required by the plant (aircraft): I_{charge}

8.1.4/ ENERGY MANAGEMENT OF THE SYSTEM

Generally, energy management must ensure the supply of the electric power demanded by the plant from the various sources. This must be done in an optimal way depending on characteristics of the sources and based on electrical constraints of the EPS. The energy management strategy thus consists in measuring in real time the current needed by the plant and in distributing it in the form of current references between sources. The strategy is based on an allocation based on the dynamics of acceptable currents by the sources. Therefore, the power source (SP), the battery (BAT), and the super-capacitor pack (SCAP) are asked to provide current with dynamics respectively slow, average and fast.

As a reminder, the objectives of the energy management system are:

- regulating the bus voltage,
- mastering the load states of the storage elements,
- limiting the power supply by the power source.

The bus voltage is regulated around a reference that can be modified by the energy management strategy, but that is by default constant (18V). As noted previously, the currents of the sources BAT, SCAP, and SP are regulated around references fixed by the energy management. The determination of these current references is done considering:

- the configuration of the EPS (depending on the available sources),
- the energy need of the plant,
- the load states of the storage elements,
- the constraints linked to the components technological limits.

8.1.5/ THE PLANT SYSTEM

The plant is a virtual model of 14 aircraft's instruments that may be activated over time by a pilot. Instruments activation (or mode) is triggered at specific times, which are given in the specification. The table 8.3 lists the requirement over the plant. Requirements 1 to 8.4 concerns the duration of each mode. For this experiment, we did not considered time as a coverage criteria. Therefore, we used these requirements to specify duration of operations during the concretization step. Requirement 9 to 17 concerns modes activation order.

In addition to the previous requirements, the domain experts provided us a matrix of instrument activation in function of the actual mode. The Table 8.3 summarizes the overall matrix. For confidentiality reasons, we have removed some details concerning the activation of the instruments. For instance, a ✓ may means that an instrument is activated before being in a steady-state.

Each instrument was specified in term of their energy request over time. The domain experts provided profile of each instrument in a matrix.

Table 8.2: Plant Requirements

ID	Description
Req 1	Mode 1 shall last 15 minutes.
Req 2	Mode 2 shall last 5 minutes.
Req 3	Mode 3 shall last 45 seconds to 1 minute.
Req 4	Mode 4 shall last 45 seconds to 1 minute.
Req 5	Mode 5 shall last 10 minutes.
Req 6	Mode 6 shall last 1 minute.
Req 7	Mode 7a, 7b, 7c, and 7d are included in the Mode 7 and the Mode 7 shall last 3 hours and 30 minutes.
Req 7.1	Mode 7a shall last 5 minutes.
Req 7.2	Mode 7b shall last 50 minutes.
Req 7.3	Mode 7c shall last 2 hours and 30 minutes.
Req 7.4	Mode 7d shall last 5 minutes.
Req 8	Mode 8a, 8b, 8c, and 8d are included in the Mode 8 and the Mode 8 shall last 3 hours and 30 minutes.
Req 8.1	Mode 8a shall last 5 minutes.
Req 8.2	Mode 8b shall last 50 minutes.
Req 8.3	Mode 8c shall last 2 hours and 30 minutes.
Req 8.4	Mode 8d shall last 5 minutes.
Req 9	Each scenario shall start with Mode 1 followed by Mode 2.
Req 10	Mode 2 shall be followed by the Mode 3 or the Mode 4.
Req 11	Mode 3 and Mode 4 shall be followed by the Mode 5.
Req 12	Mode 5 shall be followed by the Mode 6.
Req 13	Mode 6 may be repeated.
Req 13.1	Mode 6 is repeated if and only if Mode 6 is inserted in Mode 7 or Mode 8. First example: 6, 7a, 7b, 6, 7c, 7d - Second example: 6, 7a, 7b, 6, 8c, 8d
Req 14	Mode 7 and Mode 8 shall be executed satisfying the order a, b, c, and d.
Req 15	Mode 7 (a, b, c, d) may be followed by Mode 8. First example: 7a, 8b, 8c, 8d - Second example: 7a, 7b, 7c, 8d
Req 16	Mode 8 may be followed by Mode 7 if and only if the Mode 9 is activated. First example: 8a, 9, 7b, 7c, 7d - Second example: 8a, 8b, 9, 7c, 7d
Req 17	Mode 9 may be repeated twice successively.

8.2/ EXPERIMENTAL MOTIVATIONS

In this section we detail the goal of the experimentation. To address this goal, we detail the protocol we used during the experimentation. Finally, we discuss potential threat to validity associated to the protocol.

8.2.1/ GOAL OF THE CASE STUDY

We used our prototype for carrying out the EPS case study in a *In-the-Loop* context. We identified two main motivations for this case study. The first concerns the process from an engineer point of view, whereas the second concerns the validation capability of our approach. Within this case study we wanted to answer the following questions:

Table 8.3: Instruments Activation in Function of the Selected Mode

Mode	I1	I2	I3	I4	I5	I6	I7	I8	I9	I10	I11	I12	I13	I14
Mode 1	✗	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
Mode 2	✗	✓	✓	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗
Mode 3	✗	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗
Mode 4	✗	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗	✓	✗
Mode 5	✓	✓	✓	✓	✓	✗	✓	✓	✗	✗	✗	✗	✗	✗
Mode 6	✓	✓	✓	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗	✓
Mode 7a	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗
Mode 7b	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✗
Mode 7c	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗	✓	✓	✗	✗
Mode 7d	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✗	✗	✗	✗
Mode 8a	✓	✓	✓	✓	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗
Mode 8b	✓	✓	✓	✓	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗
Mode 8c	✓	✓	✓	✓	✗	✓	✓	✗	✗	✗	✓	✓	✗	✗
Mode 8d	✗	✓	✓	✓	✗	✓	✓	✗	✓	✓	✗	✗	✗	✗

1. Is our combined approach efficient to perform simulation and test generation?
   - Does our combined approach raise major issues of complexity during the modelling stage?
   - In what extent our combined approach is scalable?
2. Does our combined approach enable to validate a physical system and its model at the soonest?
   - Are we able to generate continuous signal efficiently?
   - Does our approach permit to raise the confidence in the system?
   - Does our approach enable to discover scenarios that do not satisfy a requirement?

8.2.2/ EXPERIMENTAL PROTOCOL

To answer the questions raised in the previous section, we have divided the experimentation into seven stages:

1. analysis of the EPS specification (EMR to SysML),
2. EPS modelling using SysML subset for simulation,
3. simulation of the system using a scenario example and feedback from the domain experts,
4. plant modelling: instruments of the aircraft with their energy request over time during the activation period,
5. abstraction and discretization of the plant's behaviour using the SysML subset for animation (state-machine, OCL4MBT subset, and operation with call events),

6. animating the model and generating test cases,

7. executing test cases on the simulation model and results feedback from the domain experts.

8.2.3/ THREAT TO VALIDITY

As for the previous case study, the modelling stage was made by an engineer with SysML knowledge but without any initial knowledge on EMR. Hence, the challenge concerned the interpretation of an EMR specification to design a SysML model in order to perform simulations. In this case the initial specification could influence the engineer during the SysML modelling step. In addition the engineer is familiar with model-based testing approaches. It might bias us to have an objective view of the complexity applying the proposed approach.

The second threat to validity we identified concerns the plant modelling approach. Especially, the consideration of the requirements 1 to 8.4 (see Table 8.3). Indeed, the timing constraints expressed by those requirements are not seen as safety properties (they could be as part of future work). For now, they only specify when the pilot is authorized to activate one or several instruments. Thus, this imply a strong hypothesis concerning the timing properties of the system: what happens if a wrong time activation occurs? The hypothesis is as follows: activating an instrument at a wrong time does not affect the system's current state, such that the system is considered safe regarding its timing constraints. Therefore, timing constraints are first abstracted during the system modelling stage and then used during the concretization.

The last threat to validity concerns the case study itself. We were able to use the overall approach on this case study only. Regarding the process, deeper investigations are required to provide a complete report about scalability and efficiency of the overall approach, in particular w.r.t. industrial practices on large-scale systems. In addition, this may leads to false theoretical insight concerning the unification of discrete and continuous features at a high-level of abstraction.

8.3/ MATHEMATICAL AND SYSML MODELLING

We present in this section the mathematical model of each component. The resulting SysML models are also presented.

8.3.1/ THE POWER SOURCE MODULE

The power source sub-system comprises 4 components: a power source voltage, a DC/DC buck converter, an inductor, and a current regulator with the inversion of the buck converter. They are all presented in that order.

THE SOURCE SP

The power source is a power source voltage. It is considered as an ideal voltage source, so it delivers a constant voltage V_{SP_nom} whatever the current I_{SP}, where $V_{SP\_nom} = 25V$ is the nominal voltage of the power source.

The following are the technological limits of the power source SP:

- constant voltage: $V_{SP\_nom} = 25V$
- constant maximum current: $I_{SP\_max} = 50A$
- constant maximum power: P_{SP_max} = 1250 W (under 25 V)

THE BUCK CONVERTER

The buck converter is a conversion element with a setting variable m_{buck}. The relationships between the input variables and the output variables are detailed by the Equation (1) and (2).

$$I_{SP} = m_{buck} \cdot I_{bus\_SP} \tag{1}$$

$$V_{buck} = m_{buck} \cdot V_{SP} \tag{2}$$

- I_{SP}: current in the SP source (A)
- I_{bus_SP}: current of the buck converter (from the inductor side) (A)
- V_{SP}: voltage across the power source (V)
- V_{buck}: voltage across the buck converter (from the inductor side) (V)
- m_{buck}: cyclic ratio of the buck converter, where $0 < m_{buck} < 1$

THE INDUCTOR

The state variable of the inductor, is the current I_{bus_SP} imposed in the upstream and downstream part of the inductor. The relationship that governs its behaviour is given by the Equation (3) (in integral form to display the physical causality).

$$I_{bus\_SP} = \frac{1}{L_{buck}} \cdot \int (V_{buck} - V_{bus} - R_{buck} \cdot I_{bus\_SP}) \cdot dt \tag{3}$$

- I_{bus_SP}: current of the inductor (A)
- V_{buck}: voltage across the buck converter (from the inductor side) (V)
- V_{bus}: voltage across the bus (V)
- L_{buck}: constant inductance of the coil (H), where $L_{buck} = 3.6mH$
- R_{buck}: constant resistance of the coil (Ω), where $R_{buck} = 0.124\Omega$

THE CURRENT REGULATOR

The current regulator is realized by a servo block, in this case an IP corrector.

The following relationship is obtained from the Equation (3):

$$V_{buck\_ref} = V_{bus} - K_{p\_Lb} \cdot I_{bus\_SP} + K_{p\_Lb} \cdot \int (K_{i\_Lb} \cdot I_{bus\_SP\_ref - I_{bus\_SP}}) \cdot dt \tag{4}$$

- K_{p_Lb}: constant proportional coefficient of the current corrector, where $K_{p\_Lb} = 2.2$,
- K_{i_Lb}: constant integral coefficient of the current corrector, where $K_{i\_Lb} = 40.1$,
- $I_{bus\_SP\_ref}$: current reference of the power source.

The $I_{bus\_SP\_ref}$ reference is scalable in time and is determined by the energy management strategy..

The cyclic ratio m_{buck} is obtained from the inversion of the Equation (2):

$$m_{buck} = \frac{V_{buck\_ref}}{V_{SP}} \tag{5}$$

SYSML MODELLING OF THE POWER SOURCE

The power source sub-system is represented by the BDD of the Fig 8.4 by the block `PowerSource`. It contains the `SourceSP`, the `Inductor`, the `BuckConverter`, the `CurrentRegulator` with its `BuckInversion`.

Note that the `Inductor` and the `CurrentRegulator` contain two additional flow ports y and u. These ports enables to send the terms that are under the integral of the Equation (3) to an integrator. The SysML block of the integrator is illustrated in Fig. B.8 (Appendix B). This integrator contains two parameters k and y_{start}, that may be locally modified when instantiating the integrator. The k parameter is the gain of the integrator, and y_{start} enables to specify an initial value for integral value. The constraint of the integrator enables to numerically calculate the integral of the received u signal. The block of the inductor and the integrator are linked in the IBD depicted in Fig 8.5. In addition, the constraint $0 < m_{buck} < 1$ (see Sect. 8.3.1) is satisfied by the block `Saturation` of the Fig B.8 (Appendix B). The `Saturation` block permits to give lower and upper limits to a value.

Each flow port is typed with connector, i.e. SysML block profiled with the `<<modelicaConnector>>`. Figure B.9 (Appendix B) shows the connectors used in the SysML model.

We have considered the technological limits of the source SP (see Sect. 8.3.1). These information are specified in the block `SourceSP` of the Fig. 8.4. In addition, to know the state of the power source regarding its technological limits, we have specified a state machine, which is depicted in Fig. 8.6. The source SP, is in an error state when the I_{SP_max} limit or the P_{SP_max} limit is exceeded.

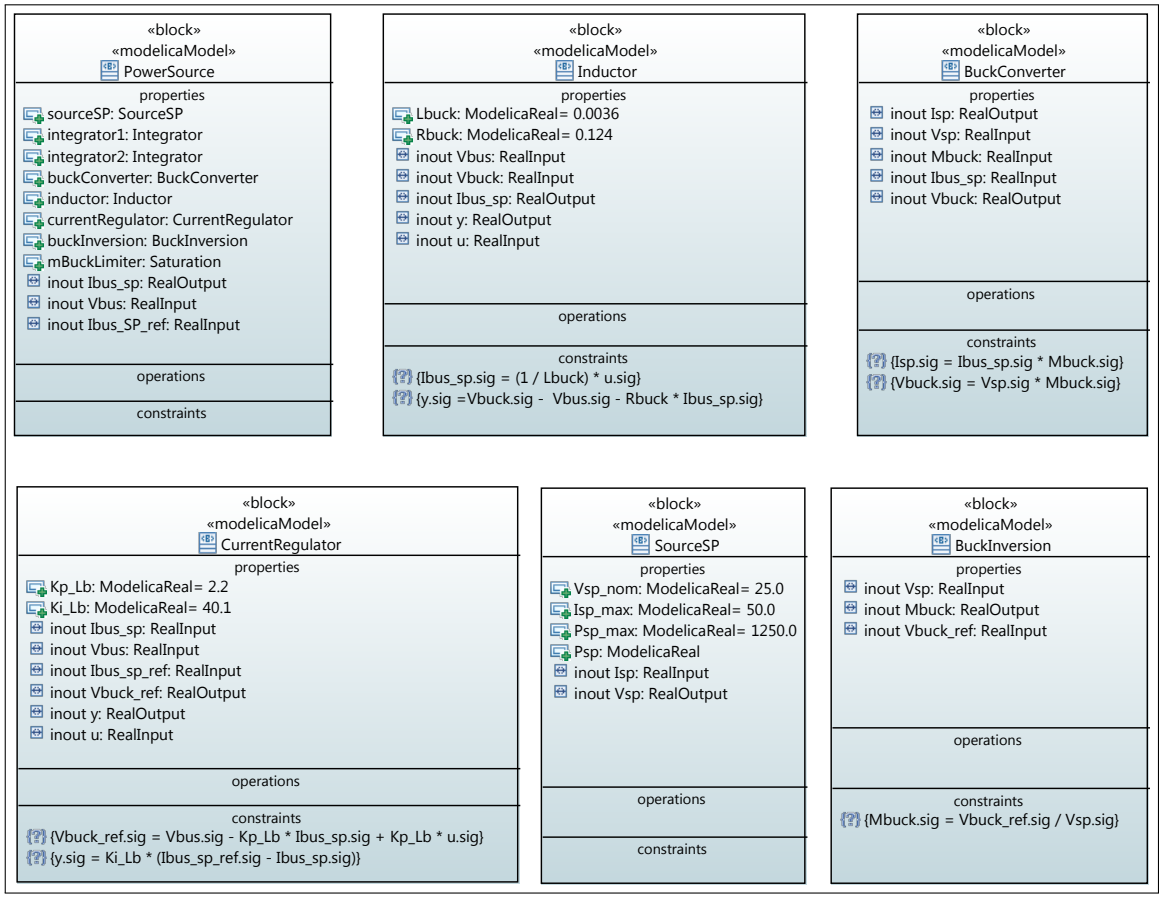


Figure 8.4: SysML BDD of the Power Source

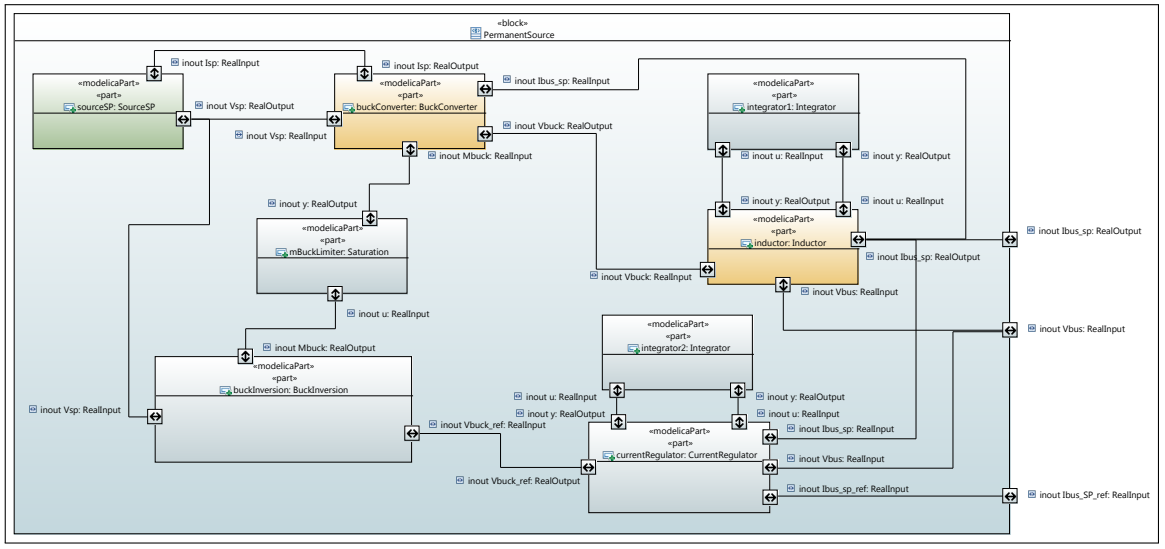


Figure 8.5: SysML IBD of the Power Source

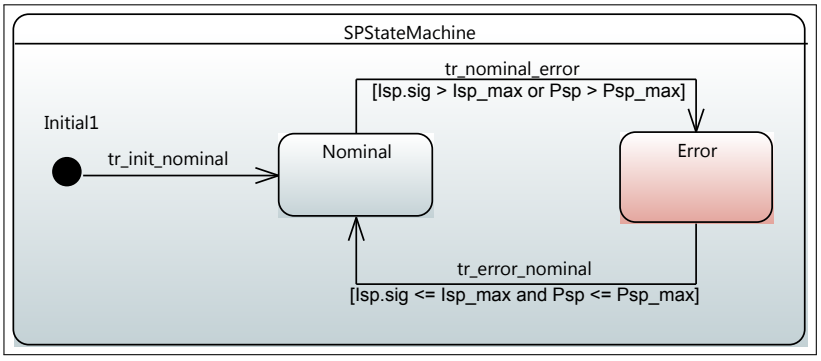


Figure 8.6: State Machine of the power source

8.3.2/ THE BATTERY MODULE

The battery sub-system comprises 4 components: an electrical voltage source, an inductor, a DC/DC boost converter, and a current regulator with the inversion of the boost converter. They are all presented in that order.

THE VOLTAGE SOURCE

The battery source is represented by an electrical voltage source. The model is a quasi-static model. The model parameters are an electromotive force or OCV (Open-Circuit Voltage) in series with an internal resistance R_{ACC}.

The voltage of the battery terminals is given by the following relationship:

$$V_{BAT} = OCV - R_{BAT} \cdot I_{BAT} \tag{6}$$

The main characteristics of this battery are: Pb-acid technology (VRLA), a capacity of 24 Ah, a nominal voltage of 12V, and an open circuit voltage of 12,6V (for $SOC = 100\%$). The state-of-charge SOC is determined by a Ah-counting method, whose Equation is given below:

$$SOC_{BAT} = SOC_0 - \frac{1}{C_n} \cdot \int I_{BAT} \cdot dt \tag{7}$$

- I_{BAT}: battery current (A),
- V_{BAT}: voltage at the source terminals (V),
- SOC_{BAT}: battery state-of-charge (%),
- R_{BAT}: constant battery internal resistance (Ω), where $R_{BAT} = 0.0475\Omega$,
- C_n: nominal capacity of the battery (Ah), where $C_n = 24x3600As$,
- OCV: battery open circuit voltage (V), where $OCV = f(SOC_{BAT})$,
- SOC_0: initial state-of-charge of the battery (%).

The evolution of the open circuit voltage OCV according to the state-of-charge SOC_{BAT} is:

$$OCV = -0.001 \cdot {SOC_{BAT}}^2 + 0.19 \cdot SOC_{BAT} + 3.55 \tag{8}$$

The quasi-static model of storage battery is produced according to the sign convention below:

- $I_{BAT} > 0$: discharge of the battery,
- $I_{BAT} < 0$: recharge of the battery.

Finally, the technological limits of the battery are as follows:

- constant maximum voltage: $V_{BAT\_max} = 12.6V$,
- constant maximum current: $I_{BAT\_max\_disch} = 24A$. The maximum current specified by the manufacturer is 240 A (10C), which corresponds to a discharge for 6 minutes. For our application, we retain a maximum current of 24 A, corresponding to a discharge for 1 hour.

THE INDUCTOR

The inductor is a storage element that imposes the current (state variable). The expression for the current is given in integral form, in the following relationship:

$$I_{BAT} = \frac{1}{L_{boost}} \cdot \int (V_{BAT} - V_{boost} - R_{boost} \cdot I_{BAT}) \cdot dt \tag{9}$$

- I_{BAT}: battery current (A),
- V_{BAT}: voltage at the source terminals (V),
- V_{boost}: voltage at the boost converter terminals (V),
- L_{boost}: constant boost converter inductance (H), where $L_{boost} = 500\mu H$,
- R_{boost}: constant resistance of the boost converter (Ω), where $R_{boost} = 0.016\Omega$.

THE BOOST CONVERTER

The boost converter is a conversion element with a setting variable m_{boost}. The relationships between the input variables and the output variables are detailed by the Equations (10) and (11).

$$I_{bus\_BAT} = (1 - m_{boost}) \cdot I_{BAT} \tag{10}$$

$$V_{boost} = (1 - m_{boost}) \cdot V_{bus} \tag{11}$$

- I_{BAT}: battery current (A),
- I_{bus_BAT}: current of the boost converter (from the bus side) (A)
- V_{boost}: voltage across the boost converter (from the source side) (V)
- V_{bus}: voltage across the bus (V)
- m_{boost}: cyclic ratio of the boost converter, where $0 < m_{boost} < 1$

THE CURRENT REGULATOR

The current regulator is realized by a servo block, in this case an IP corrector. The following Equation is obtained from the Equation (9):

$$V_{boost\_ref} = V_{BAT} - K_{p\_La} \cdot I_{BAT} + K_{p\_La} \cdot \int (K_{i\_La} \cdot I_{BAT\_ref - I_{BAT}}) \cdot dt \tag{12}$$

Regarding the inversion of the boost converter, the reference current (source side) is determined from Equation (10) and the cyclic ratio m_{boost} is obtained by inverting Equation (11).

$$I_{BAT\_ref} = \frac{1}{(1 - m_{boost})} \cdot I_{bus\_BAT\_ref} \tag{13}$$

$$m_{boost} = 1 - \frac{V_{boost\_ref}}{V_{bus}} \tag{14}$$

- K_{p_La}: constant proportional coefficient of the current regulator, where $K_{p\_La} = 2.2$,
- K_{i_La}: constant integral coefficient of the current corrector, where $K_{i\_La} = 37.2$,
- $I_{bus\_BAT\_ref}$: current reference of battery.

The $I_{bus\_BAT\_ref}$ reference is scalable in time and is determined by the energy management strategy.

SYSML MODELLING OF THE BATTERY

The block definition diagram of the battery sub-system is depicted in Fig. 8.7 and the internal block diagram is depicted in Fig. 8.8. As for m_{buck}, the constraint on m_{booost} is satisfied by a `Saturation` block. The technological limits of the voltage source are specified using the state machine depicted in Fig. 8.9.

8.3.3/ THE SUPER-CAPACITOR MODULE

The super-capacitor sub-system comprises 4 components: the SCAP source, an inductor, a DC/DC boost converter, and a current regulator with the inversion of the boost converter.

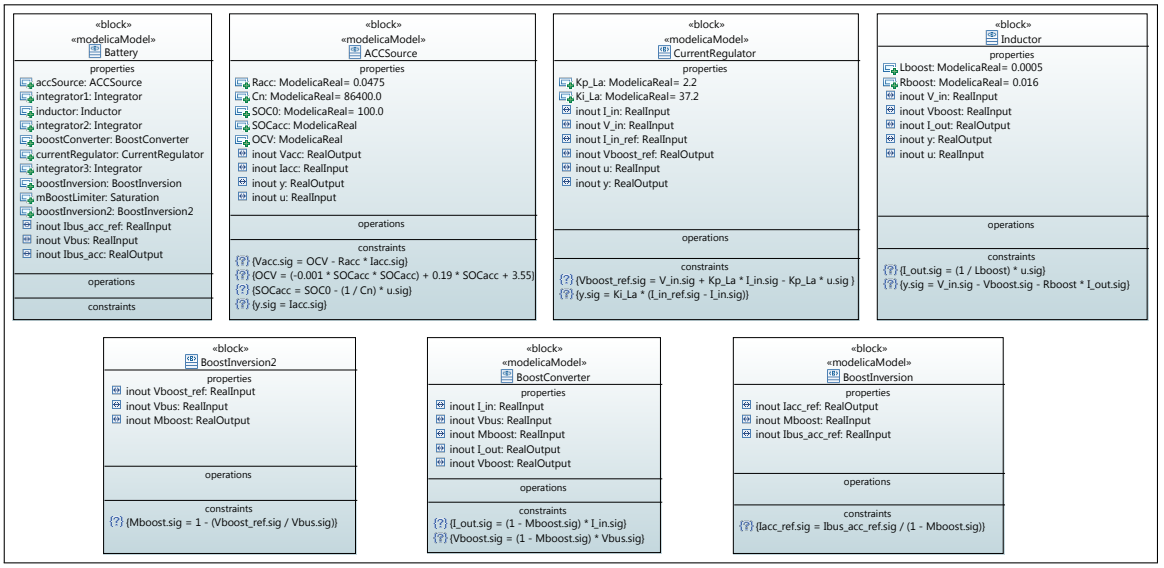


Figure 8.7: SysML BDD of the Battery

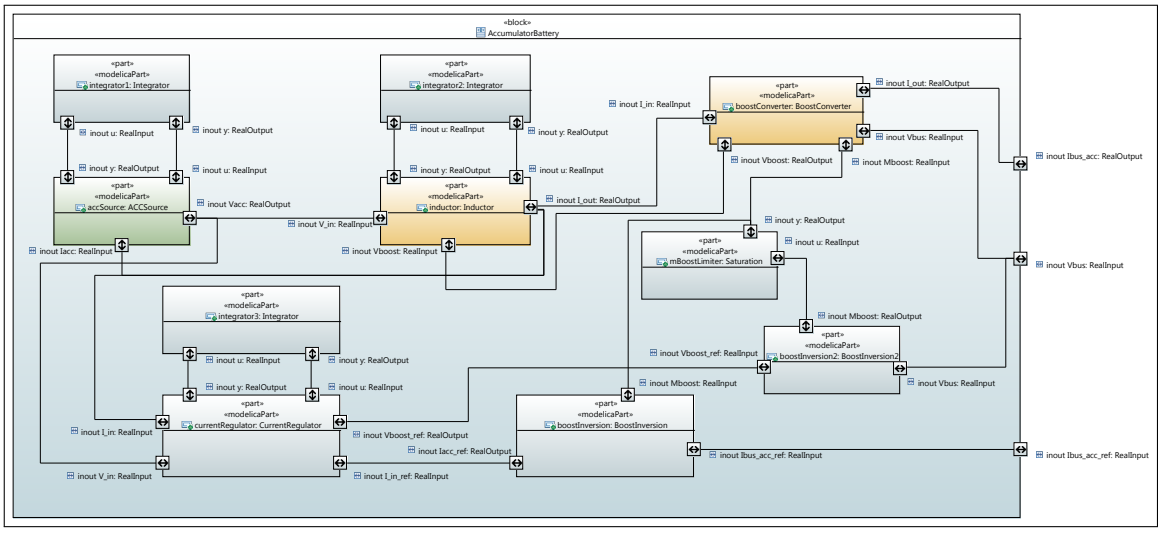


Figure 8.8: SysML IBD of the Battery

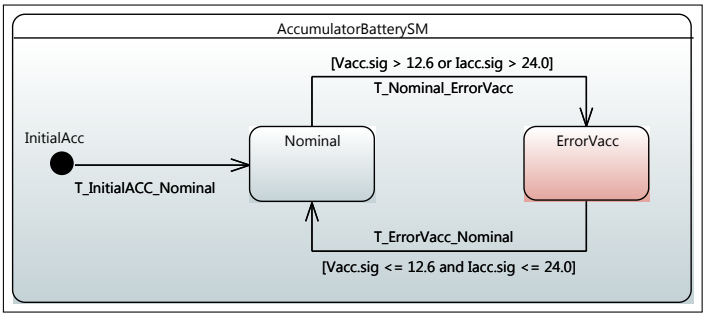


Figure 8.9: State machine of the Battery

THE SCAP SOURCE

The SCAP source is represented by an electrical voltage source. The model of the super-capacitor is obtained from a standard improved model of a super-capacitor cell. The parameters of this model are a capacity (may vary depending on the voltage across its terminals) in series with an internal resistance R_{SC}. The voltage across the super-capacitor cell is given by the following relationship:

$$V_{SC} = V_{ci} + R_{SC} \cdot I_{SC} \tag{15}$$

The voltage V_{ci} is the voltage across the equivalent capacitor. It is determined as follows:

$$V_{ci} = \int \frac{1}{C_{eq}} \cdot I_{SC} \cdot dt \tag{16}$$

$$C_{eq} = C_{i0} + C_{i1} = C_{i0} + K_{ci} \cdot V_{ci} \tag{17}$$

The super-capacitor module consists of one branch of 6 cells in series (1P6S), each cell having a capacity of 650F. We then have the following relationships:

$$I_{SCAP} = N_p \cdot I_{SC} \tag{18}$$

$$V_{SCAP} = N_s \cdot V_{SC} \tag{19}$$

- I_{SCAP}: super-capacitor module current (A),
- I_{SC}: current of one super-capacitor cell (A),
- V_{SCAP}: voltage across the super-capacitor cell (V),
- V_{SC}: voltage across one super-capacitor cell (V),
- R_{SC}: constant internal cell resistance (Ω), where $R_{SC} = 1.2m\Omega$,
- C_{i0}: constant capacity independent of the voltage (F), where $C_{i0} = 570F$,
- C_{i1}: capacity dependent of the voltage (F), where $C_{i1} = K_{ci} \cdot V_{ci}$,
- K_{ci}: constant coefficient between capacity and voltage (F/V), where $K_{ci} = 90F/V$
- N_p: constant number of parallel branches, where $N_p = 1$,
- N_s: constant number of serial cells, where $N_s = 6$.

The state-of-charge SOC_{SC} is determined as follows:

$$SOC_{SC} = \frac{{V_{ci}}^2}{{V_{c\_max}}^2} \tag{20}$$

- SOC_{SC}: SOC of the super-capacitor module (%),

- V_{ci}: voltage of across one cell (V),
- V_{c_max}: maximum voltage across one cell (V), where $V_{c\_max} = 2.7V$.

The enhanced standard model of the super-capacitor module is realized using the sign convention below:

- $I_{SCAP} > 0$: recharge of the super-capacitor,
- $I_{SCAP} < 0$: discharge of the super-capacitor.

Note that it is therefore necessary to reverse the current direction for this source in order to maintain consistency between the sources and the energy needed by the plant. The general sign convention is the following:

- $I > 0$: the source supplies a current to the bus,
- $I < 0$: the source retrieves a current from the bus.

Finally, the technological limits of the SCAP source are as follows:

- constant minimum voltage: $V_{SCAP\_min} = 7.8V$,
- constant maximum voltage: $V_{SCAP\_max} = 16.8V$,
- constant maximum current: $I_{SCAP\_max} = 30A$.

THE INDUCTOR

As for the power source and the battery, the super-capacitor comprises an inductor. It's Equation is as below:

$$I_{SCAP} = \frac{1}{L_{boost2}} \cdot \int (V_{SCAP} - V_{boost2} - R_{boost2} \cdot I_{SCAP}) \cdot dt \tag{21}$$

- I_{SCAP}: supar-capacitor current (A),
- V_{SCAP}: voltage at the super-capacitor terminals (V),
- V_{boost2}: voltage at the boost converter terminals (V),
- L_{boost2}: constant boost converter inductance (H), where $L_{boost} = 500\mu H$,
- R_{boost2}: constant resistance of the boost converter (Ω), where $R_{boost} = 0.016\Omega$.

THE BOOST CONVERTER

The boost converter is a conversion element with a setting variable m_{boost2}. The relationships between the input variables and the output variables are detailed by the Equations (22) and (23).

$$I_{bus\_SCAP} = (1 - m_{boost2}) \cdot I_{SCAP} \tag{22}$$

$$V_{boost2} = (1 - m_{boost2}) \cdot V_{bus} \tag{23}$$

- I_{SCAP}: SCAP current (A),
- I_{bus_SCAP}: current of the boost converter (from the bus side) (A)
- V_{boost2}: voltage across the boost converter (from the source side) (V)
- V_{bus}: voltage across the bus (V)
- m_{boost2}: cyclic ratio of the boost converter, where $0 < m_{boost2} < 1$

THE CURRENT REGULATOR

The current regulator is realized by a servo block, in this case an IP corrector. The following Equation is obtained from Equation (21):

$$V_{boost2\_ref} = V_{SCAP} - K_{p\_Lc} \cdot I_{SCAP} + K_{p\_Lc} \cdot \int (K_{i\_Lc} \cdot I_{SCAP\_ref - I_{SCAP}}) \cdot dt \tag{24}$$

Regarding the reversal of the boost element, the reference current (source side) is determined from Equation (22) and the cyclic ratio is obtained by inverting Equation (23):

$$I_{SCAP\_ref} = \frac{1}{(1 - m_{boost2})} \cdot I_{bus\_SCAP\_ref} \tag{25}$$

$$m_{boost2} = 1 - \frac{V_{boost2\_ref}}{V_{bus}} \tag{26}$$

- K_{p_Lc}: constant proportional coefficient of the current regulator, where $K_{p\_Lc} = 2.2$,
- K_{i_Lc}: constant integral coefficient of the current corrector, where $K_{i\_Lc} = 37.2$,
- $I_{bus\_SCAP\_ref}$: current reference of SCAP module.

The $I_{bus\_SCAP\_ref}$ reference is a direct result of the voltage control loop of the bus.

SYSML MODELLING OF THE SUPER-CAPACITOR MODULE

The block definition diagram of the super-capacitor sub-system is available in Fig. 8.10 and the internal block diagram is depicted in Fig. 8.11. The technological limits of the SCAP source are specified by the state machine illustrated in Fig. 8.12.

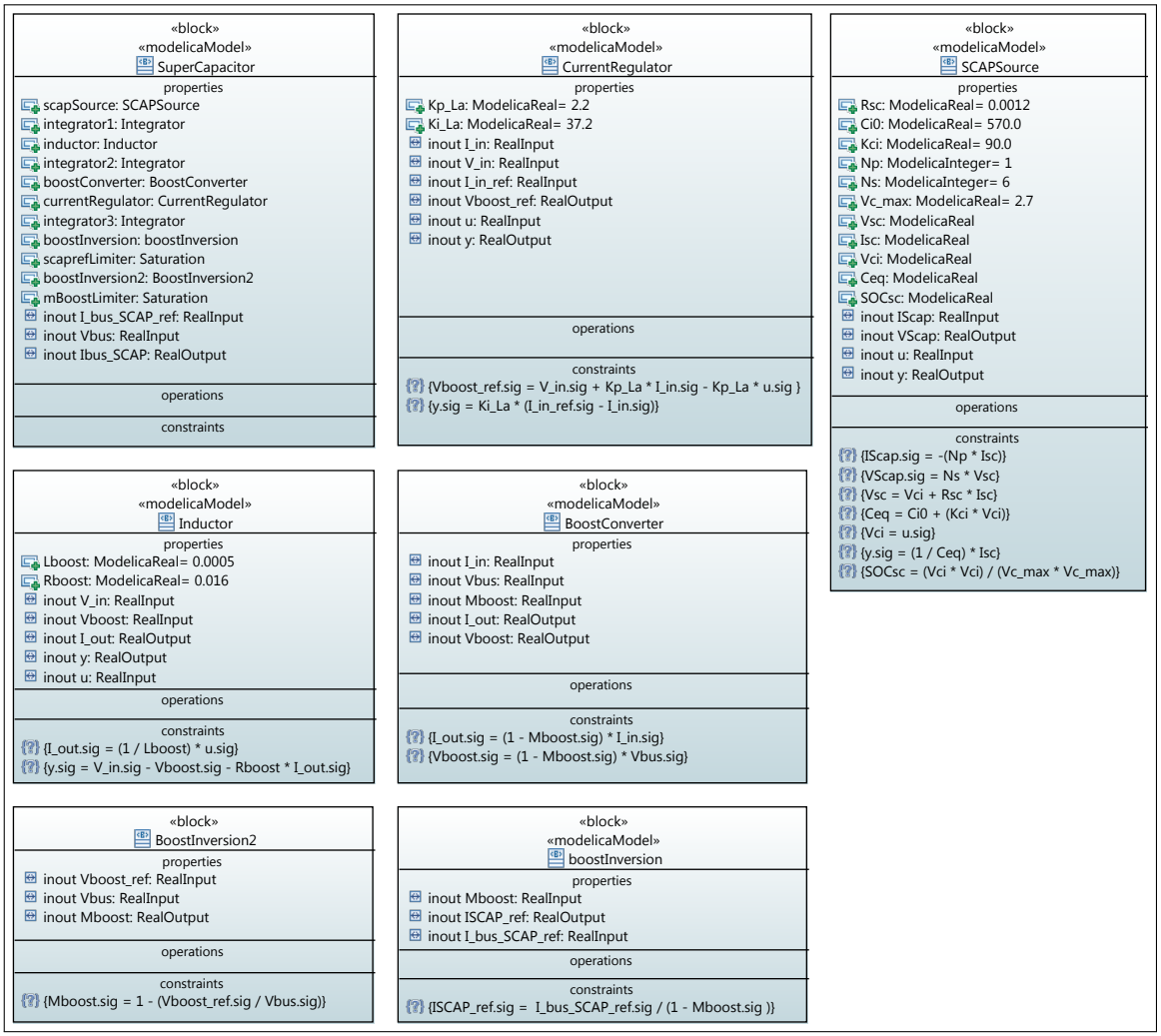


Figure 8.10: SysML BDD of the Super-Capacitor

8.3.4/ THE BUS MODULE

The bus is represented by an accumulation element (capacitor) and a coupling element. The capacitor is a storage element that imposes the voltage (state variable).

THE BUS AND ITS VOLTAGE REGULATOR

$$V_{bus} = \frac{1}{C_{Cond}} \cdot \int (I_{bus\_SCAP} - I_{bus}) \cdot dt \tag{27}$$

The electrical coupling is a current node, i.e. the sum of the currents is zero and the voltage is imposed at each instant by the bus capacitor to all connected sources.

$$I_{bus} = -(I_{charge} + I_{bus\_SP} + I_{bus\_BAT}) \tag{28}$$

Finally, the voltage regulator, is ruled by the Equation obtained from the Equation (27).

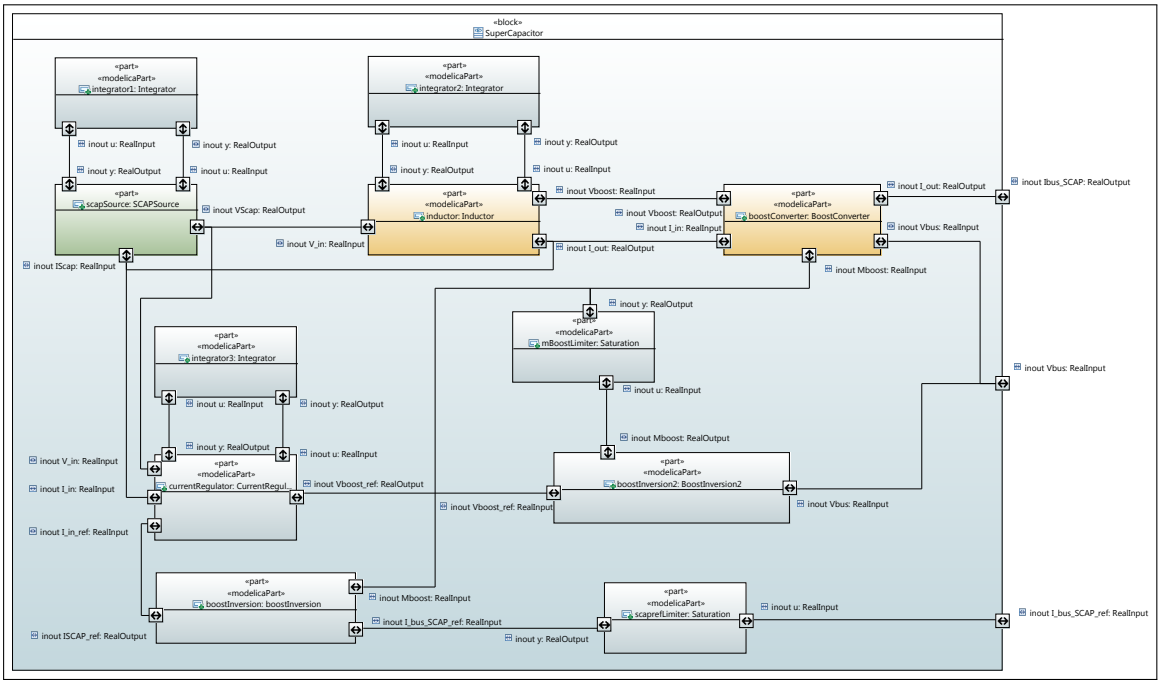


Figure 8.11: SysML IBD of the Super-Capacitor Module

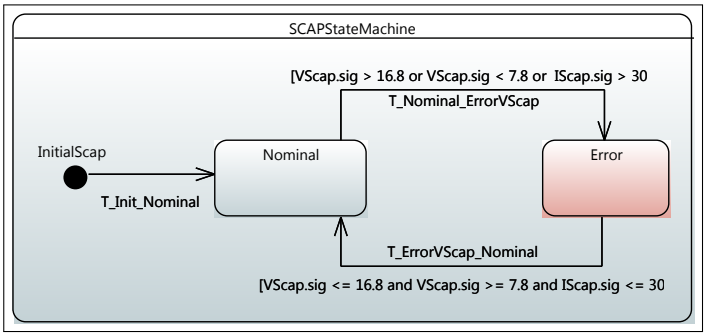


Figure 8.12: State Machine of the Super-Capacitor

$$I_{bus\_SCAP\_ref} = I_{bus} - K_{p\_c} \cdot V_{bus} + K_{p\_c} \cdot \int K_{i\_c} \cdot (V_{bus\_ref} - V_{bus}) \cdot dt \tag{29}$$

The voltage reference V_{bus_ref} is given by the energy manager, which is presented in the next section.

SysML MODELLING OF THE BUS

The block definition diagram of the bus sub-system is available in Fig. 8.13. The internal block diagram is available in Fig. 8.14.

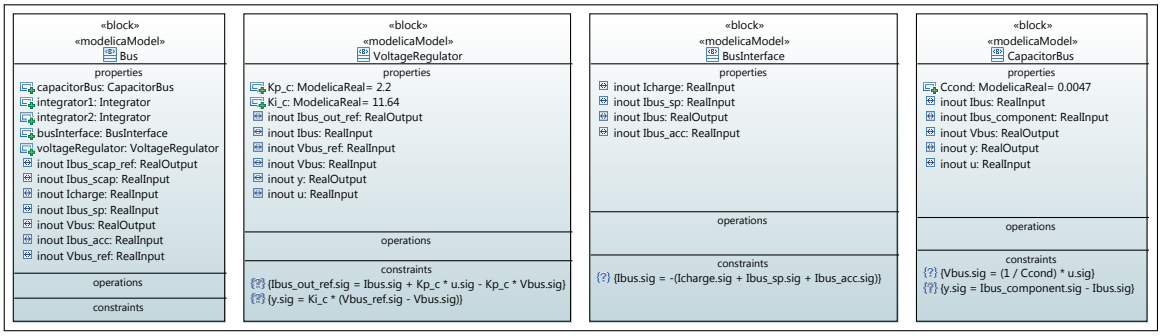


Figure 8.13: SysML BDD of Bus

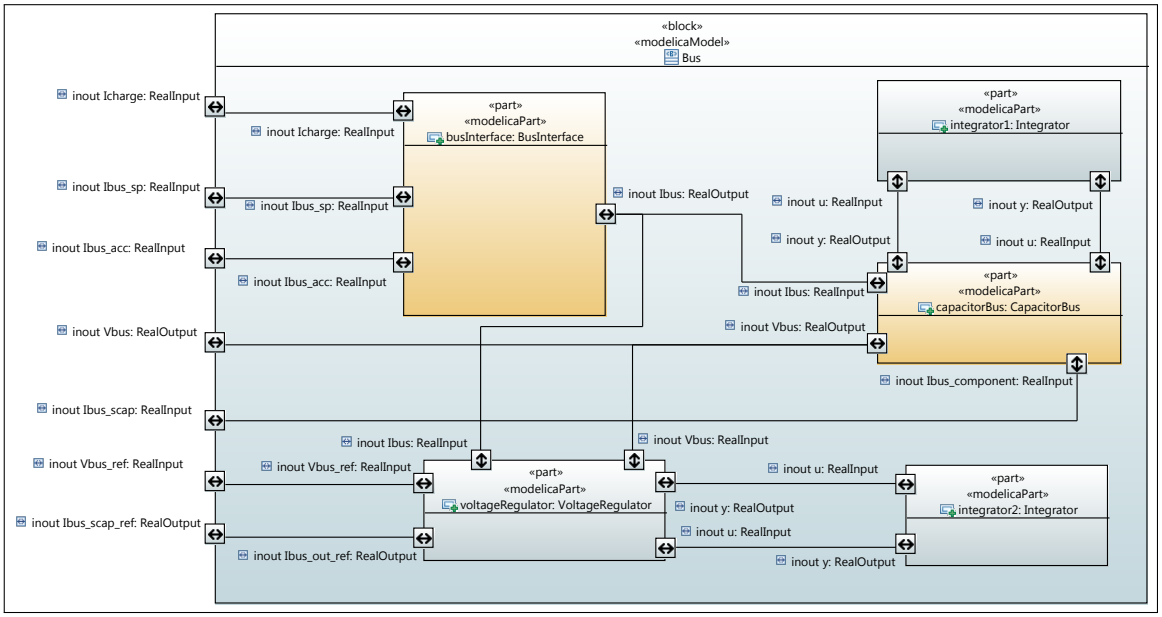


Figure 8.14: SysML IBD of the Bus

8.3.5/ THE ENERGY MANAGER MODULE

This section describes the energy manager that determines the references (voltage and current) to guarantee the power management defined in Sect 8.1.4.

The energy management strategy consists in measuring the current needed by the plant, and distributing it between sources. The strategy is based on an allocation regarding the current dynamic of the sources. The dynamic of the sources is as follows:

- the power source provides a slow dynamic current,
- the battery provides an average dynamic current,
- the super-capacitor module provides a fast dynamic current.

Therefore, the goal of the energy manager is to determine in real-time the variables $I_{bus\_SP\_ref}$ and $I_{bus\_BAT\_ref}$ in function of I_{charge} (energy needed by the plant) and I_{bus_SCAP}

(energy provides by the super-capacitor). Note that $I_{bus\_SCAP\_ref}$ results of the voltage control loop of the bus (see Equation (29)). Considering the current needed by the plant (I_{charge}), this strategy is ruled by the following logic:

- subtracting the contribution of the SCAP source I_{bus_SCAP} to I_{charge},
- the remaining current is distributed to the power source and to the battery:
  - by attributing to the power source the slow current variations through a rate limiter for which a fixed ascendant and descendant slope $rate_sp = 0.1A/s$ is defined,
  - by attributing to the battery the average current variations through a rate limiter for which a fixed ascendant and descendant slope $rate_acc = 5A/s$ is defined.

Finally, V_{bus_ref} is given by the energy manager as being constant: $V_{bus\_ref} = 18V$.

The internal block diagram of the energy manager sub-system is available in Fig. 8.15. This sub-system comprises two `Sub` blocks, two `RateLimiter` blocks and a `Saturation` block. `Sub` block enables to subtract two signals. Therefore the flow port y is the result of the subtraction between the u and v flow ports such as $y = u - v$. Concerning the `RateLimiter` block, it enables to bound the variations of current, i.e. it bounds $\dot{y}$.

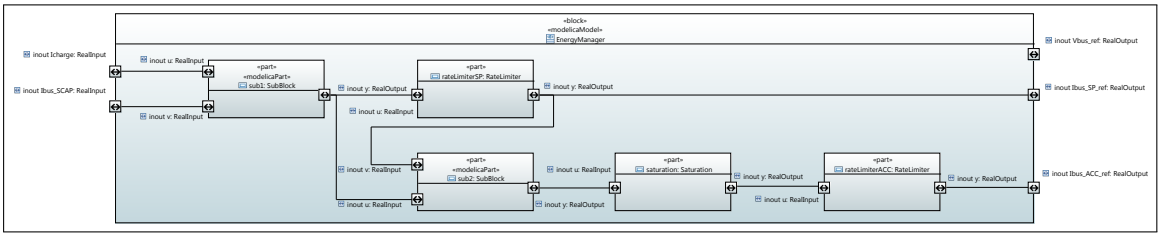


Figure 8.15: SysML IBD of the Energy Manager

The `Sub`, `Saturation`, and `RateLimiter` blocks are shown in Fig. B.8 (Appendix B).

Finally, the overall system (without) the plant is specified by the IBD of the Fig. 8.16. Each components are connected to the bus while the `energyManager` provides energy to components in function of their current variations.

The overall model presented in this section was the entry-point of Modelica code generation. Then, we performed simulations with a profile given by the domain experts. This simulation is presented in the next section.

8.4/ RESULTS OF THE EXPERIMENTATION

In order to perform simulations over the model presented in the previous section, we need to emulate energy request from the plant. At this stage, we did not have specified the plant model. Therefore, the domain experts gave us an energy profile to perform a first calibration of the model.

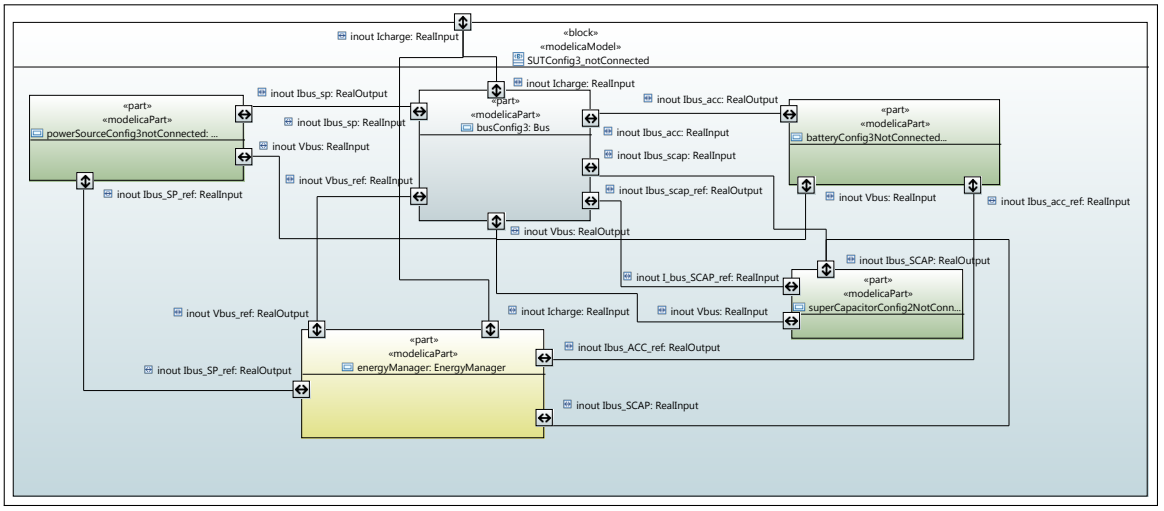


Figure 8.16: SysML IBD of System

The simulation results using the profile are presented. Then we presented the SysML model of the plant, the test generation from this model and tests execution. For confidentiality reasons, we can not provide the full model of the plant with its equations. Thus, we will rename the blocks and hide the equations. Moreover, we will show only an excerpt of the state machine that enables to perform animation and test generation.

8.4.1/ SIMULATION OF THE SYSTEM

Simulations were performed with a time step of 10ms. The profile given by the domain experts is depicted in Fig. 8.17. Note that the current is negative since it is an energy request that has to be satisfied by the electrical power system. This profile has been written as an equation in the SysML model for the `Icharge` flow port of the Fig. 8.16.

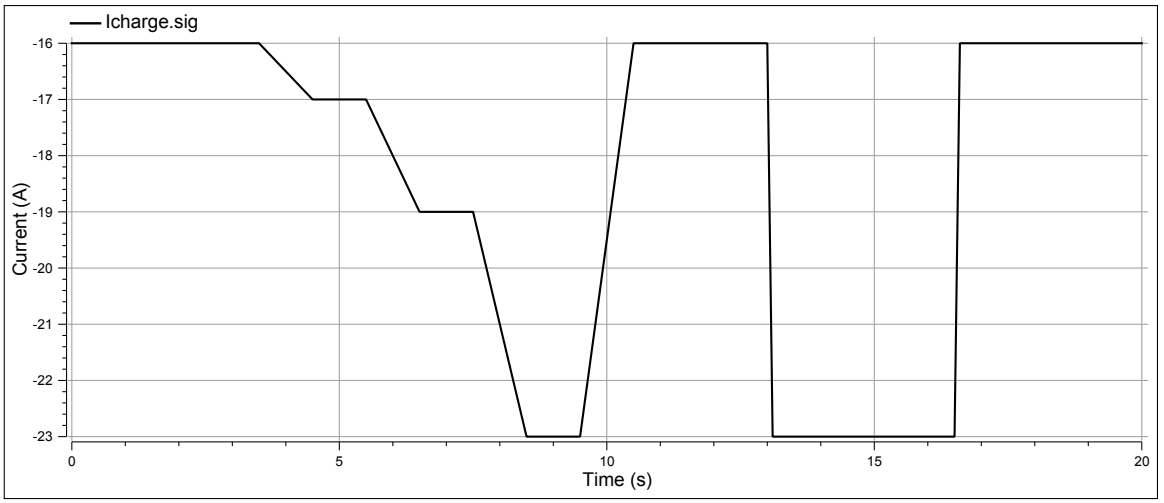


Figure 8.17: Profile Given by the Domain Experts

We observe that the sum of the currents provided by the SP source (Fig. 8.18), the BAT module (Fig. 8.19), and the SCAP module (Fig. 8.20) equals the current demanded by the plant (according to the profile given in Figure 8.17). There is also overtaking when the current's variation is very fast.

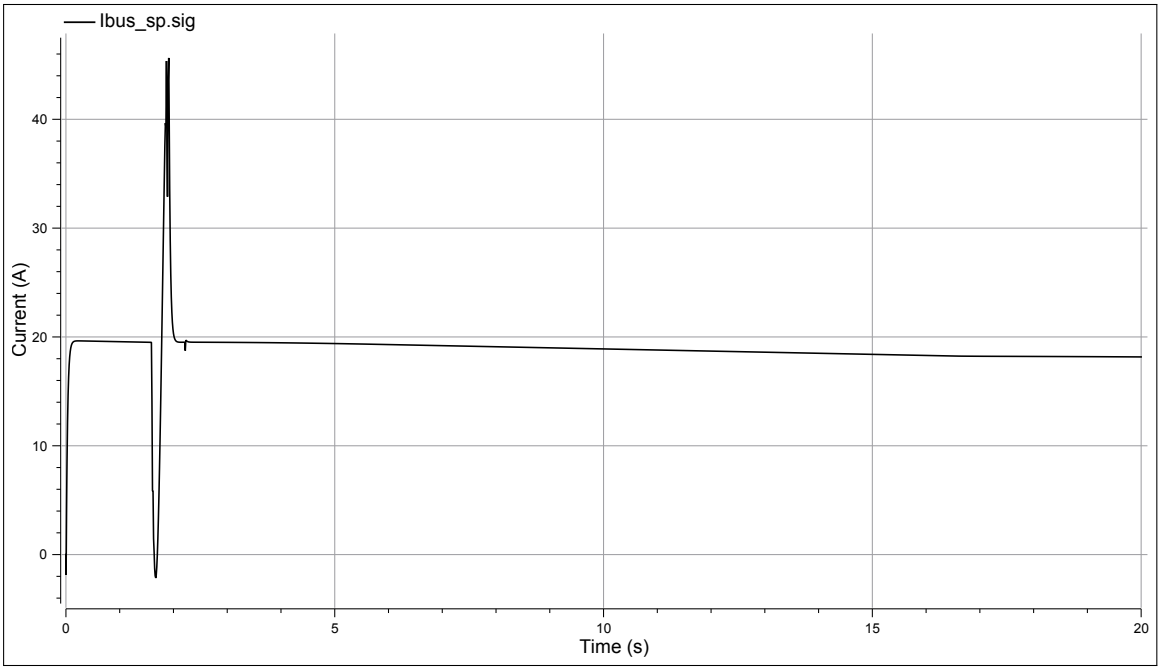


Figure 8.18: Current Provided by the Power Source Module

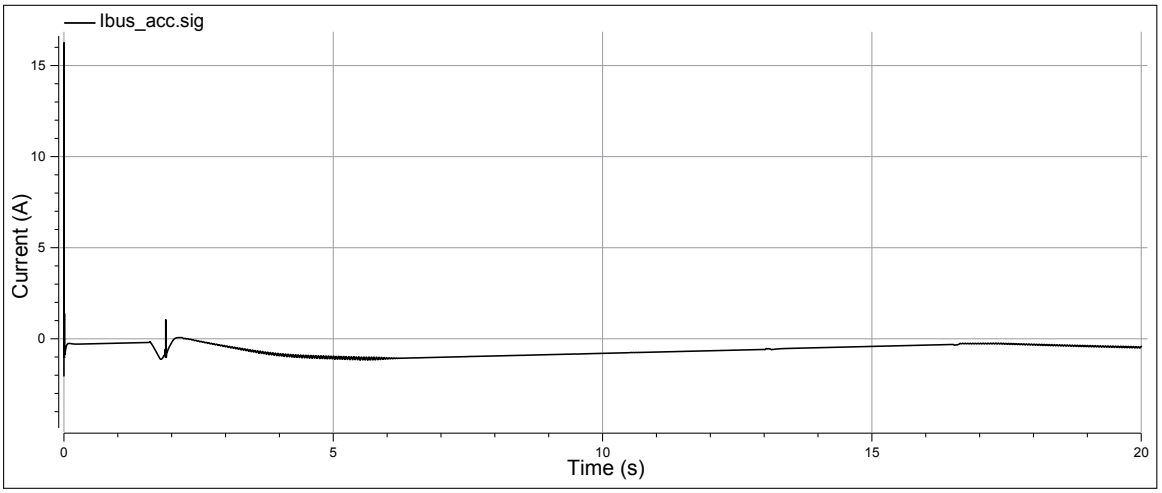


Figure 8.19: Current Provided by the BAT Module

The given profile allows us to calibrate and to validate the model. However, this profile does not represent real request from aircraft's instruments. Indeed, the plant model contains several instruments that need current for some several hour missions. The next step of this experiment concerns the automatic generation of profiles (energy requests) from the abstract and discrete view of the plant model. This last is the topic of the next section.

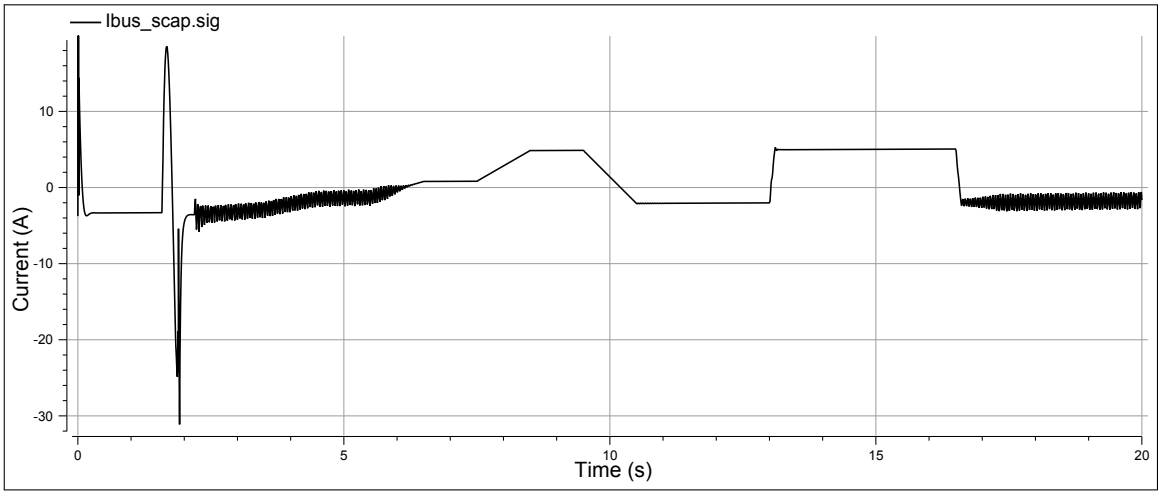


Figure 8.20: Current Provided by the SCAP Module

8.4.2/ TEST GENERATION AND EXECUTION

The model of the plant is given in Fig. 8.21 and Fig. 8.22. The plant contains 14 instruments that need energy when activated. Each instrument is connected to a bus that sum the energy request of each instrument. Then, the energy request is sent to the EPS via the `outPlant` flow port. The energy request of each instrument is given as equations over time:

```
outSignal.sig = if not isOn then 0.0 else getInstrumentNameSignal(time, relativeTime,
    timeAtActivation, timeStep);
```

The function $getInstrumentNameSignal$ computes the value of the signal as specified in the instrument specification. Then, the goal is to activate an instrument assigning its $isOn$ variable to true at the specified time.

To achieve activation triggering, we have added operations that are used as triggers ($callEvent$) in the state machine if the Fig. 8.24. Basically, a mission of the aircraft is composed of a sequence of several modes. Each mode is an activation of one or more instruments over time that are done by the pilot during the flight. The Modelica code of the Fig. 8.23 illustrates the activation of the mode 3 from the mode 2.

All the possible mode activations have been specified with a state machine. The complete state machine contains 15 states and 44 transitions. Among these 44 transitions, each of them has an event trigger, 17 have OCL4MBT guards and 18 have OCL4MBT effects. OCL4MBT enables to guide the CLPS-BZ solver during test case generation to produce sequences satisfying the requirements 9 to 17. In addition, each transition effect is completed with Modelica code in order to simulate the instruments during the continuous simulation.

The EPS and the plant were translated into a CSP using the BZP format. In this model, no operation preconditions and postconditions were used. From this model, the CLPS-BZ solver has generated 154 test cases in approximately 1 hour. The test cases were concatenated to produce the test sequences summarized in Table 8.4. These test sequences cover all the states and transitions of the state machine. In addition, the test suite covers

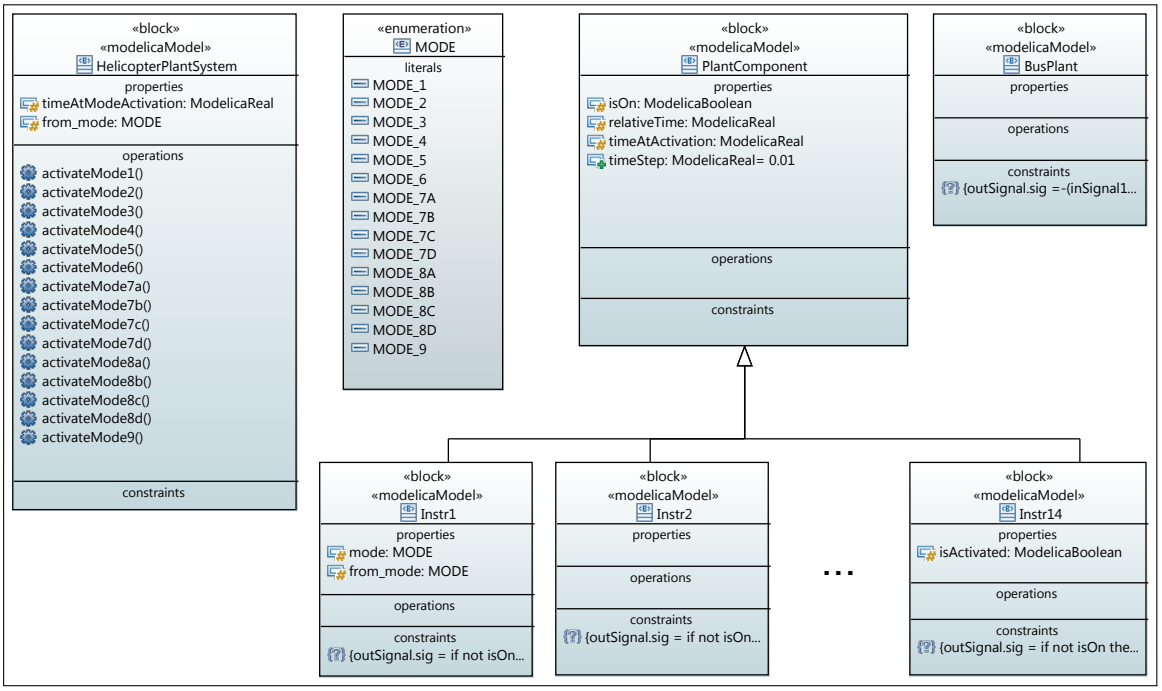


Figure 8.21: Excerpt of the Plant BDD

the requirements 9 to 17 of the Table 8.3. Each test is a sequence of 9 to 13 operation invocations.

Finally, the concretization step consisted in publishing test sequences as sequence diagrams, next translated into Modelica procedures to be simulated. For instance, the Fig 8.25 is the concretization of the first test sequence. In this thesis we performed the concretization manually. However, considering that activation times are known a priori, the automation of the concretization should be a technical issue.

A test sequence is a generated profile (current request), which represents between 4 and 5 hours of flight. We cannot provide any details on the generated profile for confidentiality reason. We have simulated several test sequences with a time step of 10ms. The simulation time for this test sequence is given in Table 8.5. For confidentiality reason, we cannot provide the simulation results of the EPS with the generated profile. However, the test suite execution permitted to observe that the BAT source fell into the error state (see the state machine of the Fig. 8.9) at multiple times.

8.5/ DISCUSSION

In this chapter, we have evaluated the proposed modelling framework with a real-life case study about an electrical power system. Thanks to these experiments, we can conclude that the proposed modelling framework, combining both discrete and continuous features of the designed system, is relevant to achieve efficient model-based testing. On the one hand, the selected SysML formalism is expressive and precise enough to describe the system, generate relevant abstract test cases, and enable early simulation of the system. On the other hand, the framework offers a concrete benefit regarding the model

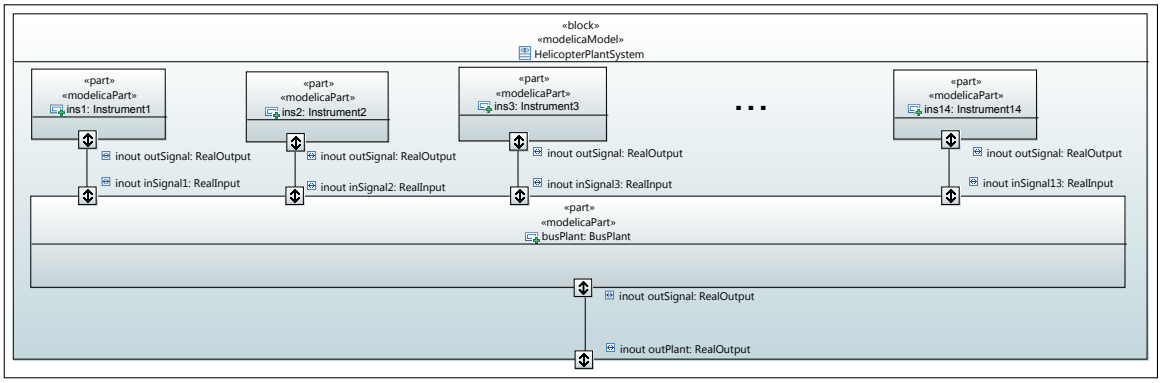


Figure 8.22: Excerpt of the Plant IBD

Table 8.4: Test Sequences after Concatenation of the Generated Test Cases

ID	Mode activation
Test 1	1, 2, 4, 5, 6, 7a, 7b, 7c, 7d
Test 2	1, 2, 4, 5, 6, 8a, 8b, 8c, 8d
Test 3	1, 2, 3, 5, 6, 7a, 7b, 7c, 7d
Test 4	1, 2, 4, 5, 6, 7a, 6, 7b, 7c, 7d
Test 5	1, 2, 4, 5, 6, 7a, 7b, 6, 7c, 7d
Test 6	1, 2, 4, 5, 6, 7a, 7b, 7c, 6, 7d
Test 7	1, 2, 4, 5, 6, 8a, 6, 8b, 8c, 8d
Test 8	1, 2, 4, 5, 6, 8a, 8b, 6, 8c, 8d
Test 9	1, 2, 4, 5, 6, 8a, 8b, 8c, 6, 8d
Test 10	1, 2, 4, 5, 6, 7a, 8b, 8c, 8d
Test 11	1, 2, 4, 5, 6, 7a, 7b, 8c, 8d
Test 12	1, 2, 4, 5, 6, 7a, 7b, 7c, 8d
Test 13	1, 2, 4, 5, 6, 8a, 9, 7b, 7c, 7d
Test 14	1, 2, 4, 5, 6, 8a, 8b, 9, 7c, 7d
Test 15	1, 2, 4, 5, 6, 8a, 8b, 8c, 9, 7d
Test 16	1, 2, 4, 5, 6, 8a, 9, 9, 7b, 7c, 7d
Test 17	1, 2, 4, 5, 6, 8a, 6, 9, 7b, 7c, 7d
Test 18	1, 2, 4, 5, 6, 8a, 8b, 6, 9, 7c, 7d
Test 19	1, 2, 4, 5, 6, 8a, 8b, 8c, 6, 9, 7d
Test 20	1, 2, 4, 5, 6, 7a, 6, 7b, 6, 7c, 6, 7d
Test 21	1, 2, 4, 5, 6, 7a, 6, 8b, 9, 9, 7c, 6, 8d
Test 22	1, 2, 4, 5, 6, 7a, 7b, 6, 8c, 6, 9, 9, 7d

writing and maintenance since the discrete and continuous features are mapped and kept consistent within the SysML model. In addition, they can be automatically checked using test case generation and simulation.

As a consequence, the supporting implementation offers a relevant execution platform for rapid prototyping and early validation. These experiments have also highlighted the high level of automation regarding test case concretization, which is known to be tricky and time-consuming, especially when real-time constraints occur, as observed in previous work [Ambert et al., 2013] where discrete model and continuous program were distinctly

```
when pre(Mode2) and activateMode3Event then /*Transition's trigger*/
 Mode2:= false;
 Mode3:= true;
 /*Transition's effect*/
 timeAtModeActivation := time;
 from_mode := MODE.MODE_2;
 ins1.from_mode := from_mode;
 ins1.mode := MODE.MODE_3;
 ins1.timeAtActivation := timeAtModeActivation ;
 ins4.isOn := true;
 ins4.timeAtActivation := timeAtModeActivation ;
 ins6.isActivated := false;
end when;
```

Figure 8.23: Generated Modelica Code of the Transition T2

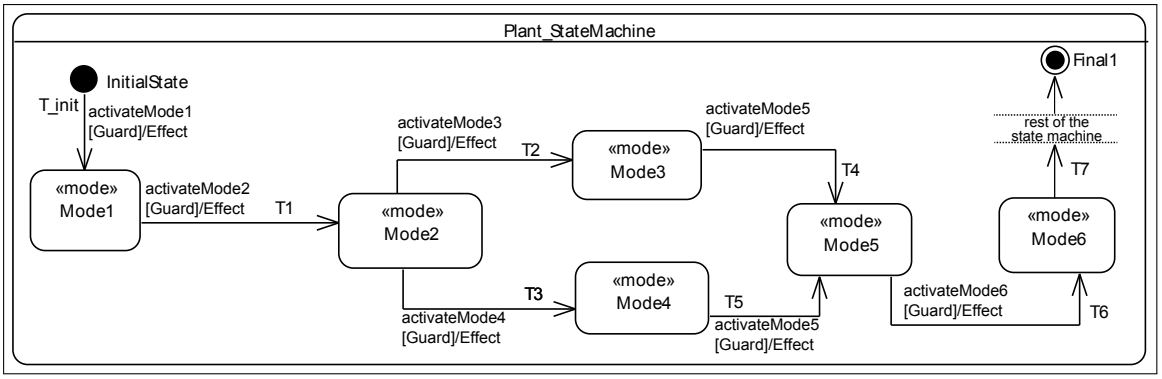


Figure 8.24: Excerpt of Plant System State Machine

Table 8.5: Simulation Time for the Test Case

Simulation step	Time (s)
Creating output file	19.8731 [25.4%]
Event-handling	1.51923 [1.9%]
Overhead	5.01971 [6.4%]
Simulation	51.9441 [66.3%]
Total	78.35614 [100.0%]

and separately managed and synchronized. This benefit stems again from the native link between discrete model elements (basis of the test generation) and the Modelica code (basis of the simulation).

From a theoretical point of view, we have the insight that a state of the system is a piecewise continuous function. A piecewise continuous function has the following properties:

- it is defined through its interval (here time),
- its constituent function are continuous on that interval,
- there is no discontinuity at each endpoint of the sub-domains within that interval.

In this case study, we noted that a state is equivalent to an activation of continuous behaviours over time. That is, a state is a timed period in which continuous variables evolve

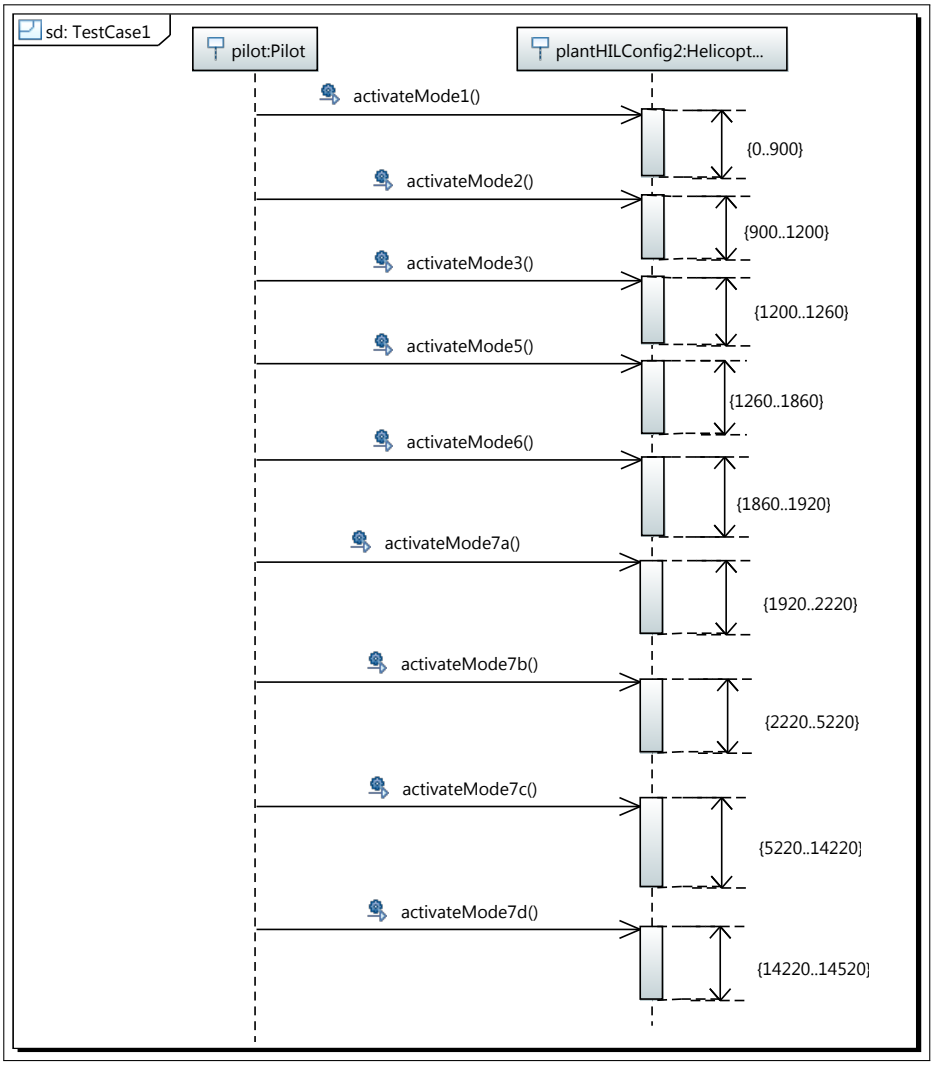


Figure 8.25: Sequence Diagram of a Generated Test Sequence

as described by the equations. Indeed, we created continuous signals from discrete animation of a state machine by triggering continuous behaviours at specific times. Thus, a state (at a high-level) may be considered as a piecewise continuous function over time.

Finally, the proposed process lack of formal semantics for time modelling. Within this case study, we did not consider time during the SysML modelling stage. Timing constraints are taken into consideration during the concretization of abstract test cases. As part of future improvements it could be interesting to specify timing constraints on transitions or operations using MARTE. In addition, we may specify the absolute and relative time of the plant's components using the MARTE profile.

IV

Conclusion and Future Work

9

CONCLUSION

This dissertation presents a unified approach for the validation of safety-critical embedded systems combining simulation and automatic test generation. The main contribution of this dissertation concerns the combination of discrete and continuous domains at a high level of abstraction and how to make this unification an asset for validation purposes.

Mixing discrete with continuous domains is not a new challenge. For instance, the DEV/DESS [Zeigler et al., 2000] formalism led to the theoretical basis of such an integration. It mainly concerns theoretical background on modelling and simulation of hybrid systems (systems with discrete and continuous aspects). In this dissertation we focused on the following issue: how to combine discrete and continuous domains to raise the confidence of safety-critical systems? From a theoretical point of view it consists in exploring the continuous states using simulation while avoiding infinite state exploration issues using abstraction and discretization. From a technical point of view, our final goal is to cleverly combine techniques of numerical solving with test generation processes in order to find execution traces that violate one or more requirements.

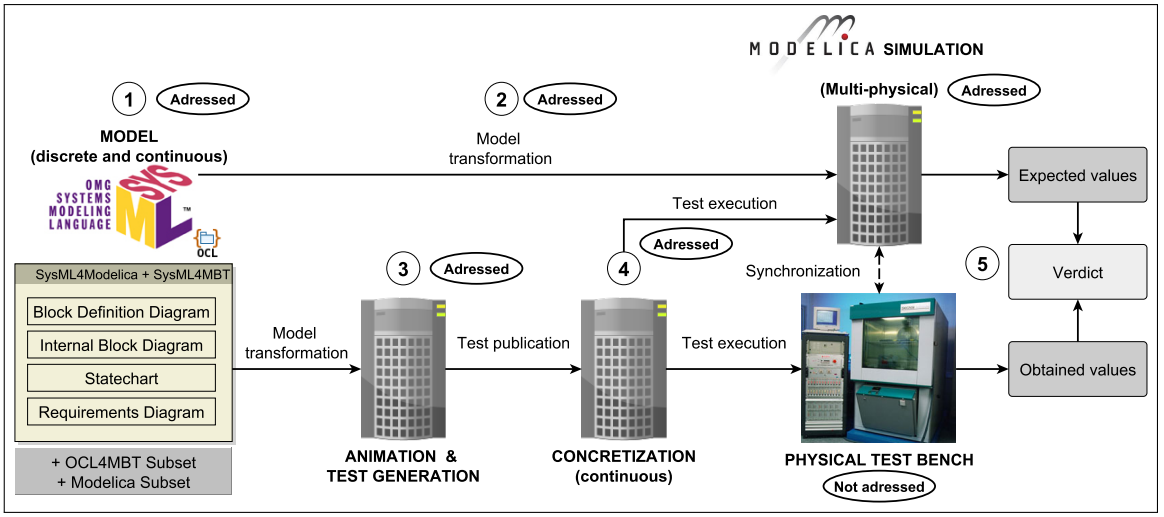


Figure 9.1: Proposed Tooled Process for Simulation and Testing

Figure 9.1 depicts the proposed tooled process we developed during the thesis. Our proposal is based on the use of model-based testing for abstract and discrete exploration of the system's states while simulation enables to explore continuous intermediate states.

It becomes possible because model-based testing generally uses constraint solvers that compute abstract and discrete models whereas numerical solvers, used for simulation, can address both continuous and discrete behaviours.

To address unified techniques for validation purpose, we proposed to combine them at a high-level of abstraction. This enables to take into account rapid prototyping and validation process at the earliest stage of design processes. Therefore, we identified three important research questions to address discrete and continuous issues at a high-level modelling stage:

- **RQ1** In what extent is it possible to build a unified model for simulation and testing?
- **RQ2** How to make executable such models?
- **RQ3** In what extent the proposed approach could be automated?

We discuss these research questions in the following sections.

9.1/ A COMBINED MODELLING APPROACH FOR SIMULATION AND TESTING

We proposed to combine continuous features with discrete features in the same SysML model. To achieve this goal, we first proposed to use VHDL-AMS to specify the continuous features within SysML [Bouquet et al., 2012, Gauthier et al., 2013]. Then, we proposed to combine the SysML4Modelica meta-model with the SysML4MBT meta-model. Results from the case studies have shown that the proposed approach is efficient to achieve early validation using simulation and test generation.

We may take a global view concerning the continuous modelling approach within SysML. The goal is to numerically resolve continuous state variables. Hence, we may consider that the most important concept concerns the definition of SysML properties as continuous variables that have to be resolved by equations. Indeed, properties variability and equations are essentials to define such continuous behaviours. In this context, the SysML4Modelica profile gives a relevant framework to build Modelica code but it could be put aside (in a first time) if the notion of variability is included in SysML.

Concerning the abstract and discrete side of the approach, we propose to use OCL and operations to lead the test generation process. In this context, operations are used to trigger continuous behaviours. Operations trigger piecewises continuous functions at specific time while the OCL code enables to formalize the behaviours of such discrete models. Hence, adding abstract and discrete semantics to continuous models for model-based testing purpose remains relatively simple and clear: operations, triggers and a subset of OCL4MBT [Gauthier et al., 2015b].

9.2/ SysML-based simulation and animation

In this thesis, we proposed to perform simulation and model animation from unified SysML models containing block definition diagrams, internal block diagrams, state diagrams, and sequence diagrams for concretization of the test sequences. The block definition diagram and the internal block diagram enable to represent the continuous part of physical systems with equations whereas the state diagram is used both to generate test sequences and to simulate the test sequences on the virtual model. We also used state diagram to represent specific states of a system such as error states (see Fig. 8.6).

Such models are therefore interpreted to enable constraints reasoning and simulation. We decided to perform model transformation and code generation using ATL and Acceleo. Hence, we have proposed a Modelica meta-model (simulation) and a BZP meta-model (animation). These meta-model are the targets of model transformations from SysML models. The overall approach is implemented in an Eclipse environment. In addition, we implemented the SysML-Modelica Transformation specification in collaboration with the OMG [Gauthier et al., 2015a].

From the experiments, we demonstrated that the proposed framework is efficient to provide simulation code and acceptance test sequences to validate safety-critical systems. Indeed, the test generation strategy is based on coverage criteria over the discrete model: state coverage, transition coverage, and decision coverage (within OCL). Hence, the proposed approach permits to generate acceptance scenarios that can be simulated to validate the model as well as the real implementation. This approach could also be used during Human-In-Loop testing where generated scenarios are executed by a pilot in a cockpit simulator for instance.

Concerning the research question 2.2 (see Sect. 1.3) we decided to use the CLPS-BZ solver since it enables to consider parallel state machines. In this thesis, we did not make test generation experiments with models including parallel state machines. However, we plan to conduct further research on this subject. In addition, the research question 2.3 is partially addressed. We proposed to unify concepts that relate to continuous and discrete solvers at the modelling level and we believe that there is a strong asset in creating collaborations between these solvers. Such a combination must be carried out precisely so that the overall technique becomes more than merely a sum of the techniques.

9.3/ Level of automation

Although we have not addressed a fully automated process (test concretization into sequence diagram is hand-made), we assess that simulation, animation, test generation, test execution, and verdict assignment may be completely automated. We have proposed to automate the generation of simulation models and constraints from a unified SysML model. The automation level is increased with the use of Model Driven Architecture techniques. Therefore, it is now possible to rely on this framework to automate the test concretization and the verdict assignment.

Technically speaking, the adaptation layer may be automatically produced by parsing the generated test cases and creating sequence diagrams. In this way, the proposed framework would be able to perform code generation from sequence diagrams. The automation of the verdict assignment is a strong asset when observers are considered during the

modelling phase. In this case, it becomes possible to generate observer components, which continuously observe test results both during the execution of test sequences on the simulation platform and on the real implementation. The verdict is computed regarding a defined delta between the oracle, provided by the simulator, and the real signal of the implementation.

9.4/ ASSESSMENT

We first assess that the proposed approach adds relatively few complexity to the initial model-based testing process of the VETESS project. Indeed, a system architect with some basics knowledge of Modelica (or Matlab/Simulink) is now able to take into account continuous behaviours at the earliest stage of the design and validation processes.

The second assessment is that the proposed approach can be integrated in MIL and HIL design processes where simulation is at the center of design and validation activities. Indeed, the proposed approach permits to generate the numerical prototype and its associated virtual plant from SysML models. It also enables to verify and validate the model using test generation activity since it helps the model calibration of the SUT during MIL-testing. Concerning HIL-testing activity, the approach enables to generate scenarios from the plant model. Scenarios are then simulated to obtain signals that stimulate the implementation under test.

Finally, the case studies permitted to experiment a validation protocol in a closed loop process. The protocol starts with the continuous modelling of the SUT and the plant (from requirements and specifications). Afterwards, we propose to add discrete behaviours over the plant in order to generate test cases using coverage criteria. The test cases are then executed on the simulation model to validate the model of the plant and the SUT (model calibration). Finally, the test cases may be reused to validate the real implementation.

10

FUTURE WORK

From the research and obtained results during this thesis, we may identify future work that concern the overall approach. First, future improvements can be done at the SysML modelling level concerning requirements traceability, reverse engineering, and time modelling. Second, these improvements lead to future work over the test generation process. Indeed, considering timing constraints at the SysML level enables to define new test generation strategies based on time and equations. Finally, this thesis lays the foundations of new research that concern smart combination of constraints solvers with numerical solvers.

10.1/ MODELLING EXTENSION

In this section we propose improvements related to the modelling stage of the approach. First, we discuss reverse engineering techniques to help the adoption of the proposed approach. Then, we propose to improve the requirements traceability. Finally, we consider several SysML extensions concerning state machines and timing constraints.

10.1.1/ REVERSE ENGINEERING AND MODEL INFERENCE

The adoption of model-based simulation and testing approaches by the industry remains a challenge. To help the adoption we may propose to provide reverse engineering and model inference techniques. Indeed, the design of a new system often relies on COTS (Commercial off-the-shelf) components since they reduce cost and maintenance. Adding reverse engineering process from such libraries to create and to infer SysML models would be a strong asset for the adoption of model-based activities.

In addition, it would be possible to perform bi-directional transformations between existing Modelica models and SysML models (the OMG's SysML-Modelica transformation specification enables bi-directional transformation). Although many models inference techniques exist, their application in industrial context remains a challenge. It requires to find techniques and models to suit every application. It opens new research topics to provide techniques and tools to infer models, and to propose and evaluate testing methods from the inferred models.

10.1.2/ REQUIREMENTS TRACEABILITY

Requirements traceability is not fully addressed in this thesis. Indeed, we observed a lack of traceability between the generated test cases and the simulation results. We made an attempt to take into account the SysML requirements diagram in order to generate annotation inside the Modelica code using the `<<satisfy>>` relationship. However, we did not create traceability links between the initial requirements and the simulation results of the test sequences.

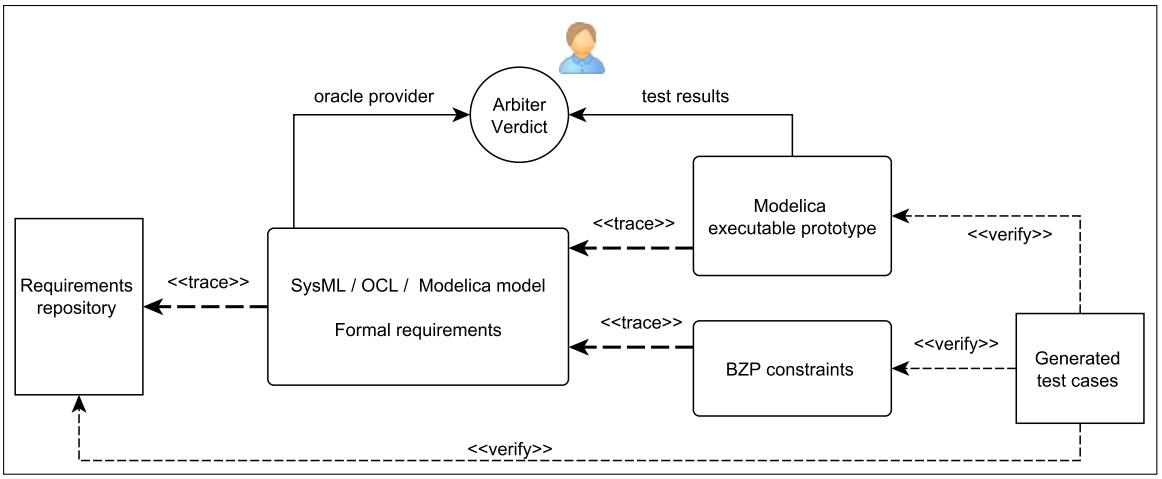


Figure 10.1: Requirement Traceability over the Proposed Approach

This is a crucial issue during the verification and the validation of the model where the initial specifications and requirements, formalized in the SysML models, define test oracles. Figure 10.1 illustrates end-to-end traceability and satisfiability between the requirements and the simulation results. In the case there is no implementation under test, requirements traceability enables to ensure that the test cases verify the requirements and that the model does not violate a requirement. To achieve this, it would be necessary to add formal properties (using OCL invariant for instance) to formalize requirements over the system's state variables. For instance, within the EMS case study, we proposed to use state machines to specify error states on state variables (see Fig 8.6). These state machines allow validation engineers to manually observe if the requirements are satisfied. The observation could be fully automated. The arbiter plays a key role because it enables to automate the satisfiability between the requirements and the simulation results.

10.1.3/ SYSML EXTENSION

We are now able to describe the continuous part of a system using equations and the discrete part using state machines. We have made experiments with simple state machines, i.e. state machines that contains an initial state, simple states and transitions. It would be interesting to generate Modelica code and test cases from more complex state machines containing several regions, choice, fork, and join states.

A potential improvement concerns the fact that no dedicated SysML4MBT entity is used to represent the time constraints. As illustrated by the EMS case study (see Chapter. 8), the inclusion of time is carried after the test generation process, during the concretization phase of the generated abstract tests.

One possible solution would be to use the MARTE UML profile dedicated to the representation of real-time embedded systems. MARTE offers all the features needed for the representation of complex time constraints. SysML and MARTE are actually complementary and several strategies can be used to efficiently combine them [Espinoza et al., 2009].

10.2/ TEST GENERATION STRATEGIES

In this section we discuss new test generation strategies based on time and equations. The previous section has presented some improvements concerning the formalization of timing constraints using the MARTE profile. These constraints can be used to define new test objectives. In addition, experiments results gave us clues to improve the exploration of the continuous state space using equations and n-wize strategy.

10.2.1/ TIME COVERAGE

Within the EMS case study, durations constraints were used to specify duration of operation during the concretization step (see the Fig. 5.7 presented in Sect. 5.4.3). It could have been possible to consider time constraints directly in the initial SysML model of the system. This would open new research perspectives about time as reachable test targets using test generation strategies based on the worth-case (or best-case) execution time (WCET, BCET) objective function. The two main criteria for evaluating a method or tool for timing analysis are safety - does it produce bounds or estimates? — and precision - are the bounds or estimates close to the exact values?

For our purpose, the time constraints may be expressed in the model at two places: either time constraints are specified in transitions guard or in operations postcondition. In the first case, a transition occurs if and only if the time constraint is satisfied. In the second case, an operation execution is valid if the time constraint is satisfied.

Typically, a timed constraint may be an interval $t = [t_1..t_2]$ or a value: $t = t_1$. We have to clarify the semantics of such time constructions. An interval specified in a transition guard means that the actual state shall last from time t_1 to time t_2 or that the system's state is allowed to change at t between t_1 and t_2. A single value specified in a transition guard means that the system's state changes at a specific value of t. From the operation postcondition point of view, the semantics is slightly different. Using interval constraints would mean that, after the execution of the operation, the time t should be between t and t_2. However, using a single value would mean that the operation lasts exactly (or finish) at $t = t_1$.

Hence, it would be possible to propose new BZP constraints over time from an abstract and discrete point of view. Computing the time elapsing from the CLPS-BZ solver is out of the question. However, depending on the above semantics, it would be possible to create new test targets based on time. For instance, considering a time interval constraint $[45..60]seconds$ on a transition's guard, it would be possible to generate several test cases to cover the constraint:

- boundary testing: t = 45s, t = 60s
- nominal testing: t = 50s
- robustness testing: t = 30s, t= 65s

From such constraints, it would also be possible to concatenate generated test cases to obtain the best/worth-case execution time of the system under test. Another possibility would be to use search algorithms over the model to determine the execution path with the best/worth execution time. A survey on time analysis such as [Wilhelm et al., 2008] would be a good starting point for future research on techniques and methods to address WCET or BCET issues in a model-based simulation and testing context.

10.2.2/ N-WISE STRATEGY

From the EMS case study, we observed that each component of the plant contains only one equation. Considering that components may have several equations, it would be possible to perform n-wize testing [Grindal et al., 2005]. Indeed, depending on human interactions, a component may react differently. It is the case in avionic and automotive systems where human interactions are simulated to study several behaviours.

For instance, consider an accelerator pedal and a steering wheel connected to a controller. One can accelerate slowly or steeply over time while turning the steering wheel in different manner. These kind of behaviours (or use cases) can be modeled as equations in the SysML model of the plant. Hence, it would be interesting to perform n-wise testing to generate relevant critical scenarios.

10.3/ COMBINING CONSTRAINT AND NUMERICAL SOLVERS

Investigating the combination of solvers to validate safety-critical systems may result two directions. The first idea is to extend CLPS-BZ to handle continuous domain in order to investigate new test generation criteria based not only on discrete features, but also on continuous ones. It would be possible to use the CLPQR library since it considers real valued variables and enables to perform linear equations solving. In the same paradigm, we may compare solvers that enable constraints and equations solving, such as the JaCoP solver [Kuchcinski and Szymanek, 2013].

The second idea consists in using separately constraint solvers and numerical solvers to provide hybrid techniques of verification and validation. We may rely on the work of [Bhadra et al., 2007] and [Ho et al., 2000] to start research of hybrid techniques for functional verification and validation. We have some insights concerning the combined use of CLPS-BZ with a numerical solver. More precisely, the use of interactive simulation, driven by a constraint solver, would enable to explore the continuous state space between two specified discrete states. This combination requires each solver to manipulate the same object, and requires establishing a communication protocol to propagate deductions made by a solver in the other. Such a protocol would not only raise issues about concurrency or synchronization but it would obviously require further investigation about more complex algorithms regarding state reachability issues, including meta-heuristics, patterns recognition, fuzzing, etc. These issues open new research topics combining parallel and distributed fields with formal verification and validation.

Hybrid techniques for functional verification and validation may be a strong asset for online testing. Online testing [Mikucionis et al., 2003, Mikucionis et al., 2004] combines test generation and execution: a single test case is generated from the model at a time and is immediately executed on the model under test and the system under test.

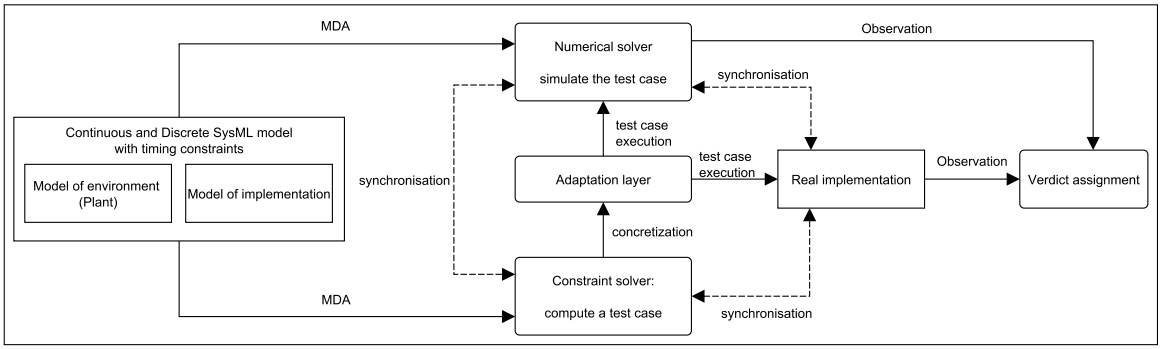


Figure 10.2: Online Testing Process Combining Numerical and Constraints solver

As depicted in the Fig. 10.2, synchronization between the solvers and the implementation under test is a key issue to achieve online testing efficiently. First, the constraint solver generates a test case according coverage criteria. The abstract test case is then automatically translated and executed on the numerical solver and on the real implementation. While the test case is executed, the solver may computes the next test case or waits for the simulation results. Indeed, simulation results could give new relevant constraints, or at least could play the role of objective function that could help the constraint solver to cover a specific critical value in the continuous domain.

BIBLIOGRAPHY

[Abrial, 1996] Abrial, J.-R. (1996). **The B-BOOK: Assigning Programs to Meanings**. Cambridge University Press. ISBN 0-5214-9619-5.

[Adrion et al., 1982] Adrion, W. R., Branstad, M. A., and Cherniavsky, J. C. (1982). **Validation, Verification, and Testing of Computer Software**. *ACM Computer Survey*, 14(2):159–192.

[Amalfitano et al., 2014] Amalfitano, D., Fasolino, A. R., Scala, S., and Tramontana, P. (2014). **Towards Automatic Model-in-the-loop Testing of Electronic Vehicle Information Centers**. In *Proceedings of the 2014 International Workshop on Long-term Industrial Collaboration on Software Engineering. (WISE '14)*, pages 9–12, Vasteras, Sweden. ACM.

[Ambert et al., 2002] Ambert, F., Bouquet, F., Chemin, S., Guenaud, S., Legeard, B., Peureux, F., Vacelet, N., and Utting, M. (2002). **BZ-TT: A Tool-Set for Test Generation from Z and B using Constraint Logic Programming**. In *Formal Approaches to Testing of Software (FATES'02)*, pages 105–120, Brnö, Czech Republic.

[Ambert et al., 2012] Ambert, F., Bouquet, F., Lasalle, J., Legeard, B., and Peureux, F. (2012). **Applying an MBT Toolchain to Automotive Embedded Systems: Case Study Reports**. In 4^{th} *International Conference on Advances in System Testing and Validation Lifecycle. (VALID'12)*, pages 139–144, Lisbon, Portugal. Think Mind Digital Library.

[Ambert et al., 2013] Ambert, F., Bouquet, F., Lasalle, J., Legeard, B., and Peureux, F. (2013). **Applying a Def-use Approach on Signal Exchange to Implement SysML Model-based Testing**. In *Proceedingf of the* 9^{th} *European Conference on Modeling Foundations and Applications. (ECMFA'13)*, volume 7949 of *LNCS*, pages 134–151, Montpellier, France. Springer Berlin Heidelberg.

[Apvrille et al., 2004] Apvrille, L., Courtiat, J. P., Lohr, C., and de Saqui-Sannes, P. (2004). **TURTLE: A Real-Time UML Profile Supported by a Formal Validation Toolkit**. *IEEE Transaction Software Engineering*, 30(7):473–487.

[Arcuri et al., 2010] Arcuri, A., Iqbal, M., and Briand, L. (2010). **Black-box System Testing of Real-time Embedded Systems Using Random and Search-based Testing**. In *Proceedings of the* 22^{nd} *International Conference on Testing Software and Systems. (ICTSS'10)*, volume 6435 of *LNCS*, pages 95–110, Natal, Brazil. Springer Berlin Heidelberg.

[Association, 2012] Association, M. (2012). **Modelica Specification - A Unified Object-Oriented Language for Systems Modeling**. https://www.modelica.org/documents/ModelicaSpec33.pdf.

[Baracchi et al., 2013] Baracchi, L., Mazzini, S., Garcia, G., Cimatti, A., and S., T. (2013). **The FOREVER Methodology: a MBSE framework for Formal Verification**. In *International Space System Engineering Conference (DASIA'13)*, Porto, Portugal.

[Beizer, 1990] Beizer, B. (1990). **Software Testing Techniques**. Van Nostrand Reinhold Co., 2^{nd} edition. ISBN 0-442-20672-0.

[Belina and Hogrefe, 1989] Belina, F., and Hogrefe, D. (1989). **The CCITT-specification and description language SDL**. *Computer Networks and ISDN Systems*, 16(4):311–341.

[Benigni and Monti, 2011] Benigni, A., and Monti, A. (2011). **Development of a Platform for Hardware in the Loop Testing of Network Controller**. In *2011 Grand Challenges on Modeling and Simulation Conference. (GCMS'11)*, pages 124–128, Hague, Netherlands. Society for Modeling And Simulation Int.

[Berthomieu and Vernadat, 2006] Berthomieu, B., and Vernadat, F. (2006). **Time Petri Nets Analysis with TINA**. In *Proceedings of the* 3^{rd} *International Conference on the Quantitative Evaluation of Systems. (QEST'06)*, pages 123–124, Riverside, CA, USA. IEEE Computer Society.

[Bézivin et al., 2003a] Bézivin, J., Breton, E., Dupé, G., and Valduriez, P. (2003a). **The ATL transformation-based model management framework**. Technical report, University of Nantes, Insititut de Recherche en Informatique de Nantes (IRIN).

[Bézivin et al., 2003b] Bézivin, J., Dupé, G., Jouault, F., Pitette, G., and Rougui, J. E. (2003b). **First experiments with the ATL model transformation language: Transforming XSLT into XQuery**. In *Proceedings of the* 18^{th} *Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. (OOPSLA'03)*, Anahelm, USA. ACM SIGPLAN.

[Bhadra et al., 2007] Bhadra, J., Abadir, M. S., Wang, L.-C., and Ray, S. (2007). **A survey of hybrid techniques for functional verification**. *IEEE Design and Test of Computers*, 24(2):112–122.

[Borba and Meira, 1993] Borba, P., and Meira, S. (1993). **From VDM specifications to functional prototypes**. *Journal of Systems and Software*, 21(3):267–278.

[Bouquet et al., 2006] Bouquet, F., Dadeau, F., and Legeard, B. (2006). **Automated Boundary Test Generation from JML Specifications**. In *Proceedings of the* 14^{th} *International Symposium on Formal Methods. (FM'06)*, volume 4085 of *LNCS*, pages 428–443, Hamilton, Canada. Springer Berlin Heidelberg.

[Bouquet et al., 2005a] Bouquet, F., Dadeau, F., Legeard, B., and Utting, M. (2005a). **JML-Testing-Tools: A Symbolic Animator for JML Specifications Using CLP**. In *Proceeding of the* 11^{th} *International Conference on Tools and Algorithms for the Construction and Analysis of Systems. (TACAS'05)*, volume 3440 of *LNCS*, pages 551–556. Springer Berlin Heidelberg, Edinburgh, UK.

[Bouquet et al., 2005b] Bouquet, F., Dadeau, F., Legeard, B., and Utting, M. (2005b). **Symbolic animation of jml specifications**. In *Proceedings of the International Symposium of Formal Methods Europe. (FM'05)*, volume 3582 of *LNCS*, pages 75–90, Newcastle, UK. Springer Berlin Heidelberg.

[Bouquet et al., 2012] Bouquet, F., Gauthier, J.-M., Hammad, A., and Peureux, F. (2012). **Transformation of SysML structure diagrams to VHDL-AMS**. In *Proceedings of the 2^{nd} IEEE Workshop on design, control and software implementation for distributed MEMS. (dMEMS'12)*, pages 74–81, Besançon, France. IEEE Computer Society.

[Bouquet et al., 2007] Bouquet, F., Grandpierre, C., Legeard, B., Peureux, F., Vacelet, N., and Utting, M. (2007). **A Subset of Precise UML for Model-based Testing**. In *3^{rd} Int. Workshop on Advances in Model-based Testing. (A-MOST'07)*, pages 95–104, London, United Kingdom. ACM.

[Bouquet et al., 2004a] Bouquet, F., Legeard, B., and Peureux, F. (2004a). **CLPS-B: A Constraint Solver to Animate a B Specification**. *International Journal on Software Tools for Technology Transfer, STTT*, 6(2):143–157.

[Bouquet et al., 2004b] Bouquet, F., Legeard, B., Peureux, F., and Torreborre, E. (2004b). **Mastering Test Generation from Smart Card Software Formal Models**. In *Revised Selected Papers of International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices. (CASSIS'04)*, volume 3362 of *LNCS*, pages 70–85. Springer Berlin Heidelberg, Marseille, France.

[Bouquet et al., 2004c] Bouquet, F., Legeard, B., Utting, M., and Vacelet, N. (2004c). **Faster Analysis of Formal Specification**. In *Proceedings of the 6^{th} International Conference on Formal Engineering Methods. (ICFEM'04)*, volume 3308 of *LNCS*, pages 239–258, Seattle, WA, United States. Springer Berlin Heidelberg.

[Bouscayrol et al., 2006] Bouscayrol, A., Lhomme, W., Delarue, P., Lemaire-Semail, B., and Aksas, S. (2006). **Hardware-in-the-loop simulation of electric vehicle traction systems using Energetic Macroscopic Representation**. In *Proceedings of the 32^{nd} Annual Conference on IEEE Industrial Electronics. (IECON'06)*, pages 5319–5324, Paris, France. IEEE.

[Boutekkouk, 2010] Boutekkouk, F. (2010). **Automatic SystemC Code Generation from UML Models at Early Stages of Systems on Chip Design**. *International Journal of Computer Applications*, 8(6):10–17.

[Braunstein et al., 2014] Braunstein, C., Peleska, J., Schulze, U., Hübner, F., Huang, W.-l., Haxthausen, A. E., and Vu, L. H. (2014). **A SysML Test Model and Test Suite for the ETCS Ceiling Speed Monitor**. Technical report, ITEA2 Project.

[Broekman, 2002] Broekman, B. M. (2002). **Testing Enbredded Software**. Addison-Wesley Longman Publishing Co., Inc. ISBN 0321159861.

[Bruel, 1996] Bruel, J.-M. (1996). **FuZE : un environnement intégré pour l'analyse formelle de logiciels distribués temps réel**. Thèse de doctorat, Université Paul Sabatier, Toulouse, France.

[Bullock et al., 2004] Bullock, D., Johnson, B., Wells, R. B., Kyte, M., and Li, Z. (2004). **Hardware-in-the-loop simulation**. *Transportation Research Part C: Emerging Technologies*, 12(1):73–89.

[Cantenot et al., 2013] Cantenot, J., Ambert, F., and Bouquet, F. (2013). **Strategies Comparison of Test Generation from UML Using SMT Solver**. In *Proceeding of the 5^{th} International Workshop on Constraints in Software Testing Verification and Analysis.*

Held in conjunction with ICST 2013. (CSTVA'13), pages 224–229, Luxemburg, Luxemburg. IEEE.

[Cantenot et al., 2012] Cantenot, J., Bouquet, F., and Ambert, F. (2012). **Transformation rules from UML4MBT meta-model to SMT meta-model for model animation**. In *Proceedings of the 12^{th} Workshop on OCL and Textual Modelling. In conjunction with Models'2012. (OCL'12)*, pages 55–60, Innsbruck, Austria. ACM.

[Carloni et al., 2006] Carloni, L. P., Passerone, R., Pinto, A., and Angiovanni-Vincentelli, A. L. (2006). **Languages and Tools for Hybrid Systems Design**. *Found. Trends Electron. Des. Autom.*, 1(1-2):1–193.

[Carr et al., 2004] Carr, C. T., McGinnity, T. M., and McDaid, L. J. (2004). **Integration of UML and VHDL-AMS for Analogue System Modeling**. *Formal aspects of computing*, 16(1):80–94.

[Cheng and Krishnakumar, 1993] Cheng, K. T., and Krishnakumar, A. S. (1993). **Automatic Functional Test Generation Using the Extended Finite State Machine Model**. In *Proceedings of the 30^{th} International Design Automation Conference. (DAC'93)*, pages 86–91, Dallas, Texas, USA. ACM.

[Chiang et al., 2010] Chiang, H.-H., Wu, S.-J., Pemg, J.-W., Wu, B.-F., and Lee, T.-T. (2010). **The Human-in-the-Loop Design Approach to the Longitudinal Automation System for an Intelligent Vehicle**. *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, 40(4):708–720.

[Clarke et al., 2002] Clarke, D., Jéron, T., Rusu, V., and Zinovieva, E. (2002). **STG: a Symbolic Test Generation Tool**. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems. (TACAS'02)*, volume 2280 of *LNCS*, pages 470–475, Grenoble, France. Springer Berlin Heidelberg.

[Clarke and Emerson, 1982] Clarke, E. M., and Emerson, E. A. (1982). **Design and synthesis of synchronization skeletons using branching time temporal logic**. In *Logics of Programs*, volume 5000 of *LNCS*, pages 52–71. Springer Berlin Heidelberg.

[Clarke et al., 1999] Clarke, E. M., Grumberg, O., and Peled, D. (1999). **Model-Checking**. MIT Press. ISBN 9780262032704.

[Conte and Boor, 1980] Conte, S. D., and Boor, C. (1980). **Elementary Numerical Analysis: An Algorithmic Approach**. McGraw-Hill Higher Education, 3^{rd} edition. ISBN: 0070124477.

[Dadeau and Tissot, 2009] Dadeau, F., and Tissot, R. (2009). **jSynoPSys - A Scenario-Based Testing tool based on the symbolic animation of B machines**. *Electronic Notes in Theoretical Computer Science*, 253(2):117–132.

[Denise et al., 2012] Denise, A., Gaudel, M., Gouraud, S., Lassaigne, R., Oudinet, J., and Peyronnet, S. (2012). **Coverage-Biased Random Exploration of Large Models and Application to Testing**. *STTT, International Journal on Software Tools for Technology Transfer*, 14(1):73–93.

[Dijkstra, 1972] Dijkstra, E. W. (1972). **Structured Programming**. Academic Press Ltd. ISBN 0-12-200550-3.

[Duchene, 2014] Duchene, F. (2014). **Detection of Web Vulnerabilities via Model Inference assisted Evolutionary Fuzzing**. PhD thesis, Grenoble University.

[Duran and Ntafos, 1984] Duran, J., and Ntafos, S. (1984). **An evaluation of random testing**. *The journal IEEE Transactions on Software Engineering*, 10(4):438–444.

[Espinoza et al., 2009] Espinoza, H., Cancila, D., Selic, B., and Gérard, S. (2009). **Challenges in Combining SysML and MARTE for Model-Based Design of Embedded Systems**. In *Proceedings of the* 5^{th} *European Conference on Model Driven Architecture - Foundations and Applications. (ECMDA-FA'09)*, volume 5562 of *LNCS*, pages 98–113, Enschede, The Netherlands. Springer Berlin Heidelberg.

[Farooq et al., 2010] Farooq, M., Adhikari, S., Haase, J., and Grimm, C. (2010). **Modeling Methodology in SystemC-AMS for Embedded Analog Mixed Signal Systems**. In *Proceedings of the* 8^{th} *International Conference on Frontiers of Information Technology. (FIT '10)*, pages 27:1–27:6, Islamabad, Pakistan. ACM.

[Ferreira et al., 1999] Ferreira, J. A., De Oliveira, J. E., and Costa, V. A. (1999). **Modeling of hydraulic systems for hardware-in-the-loop simulation: A methodology proposal**. In *Proceedings of the International Mechanical Engineering Congress and Exposition. (IMECE'99)*, Nashville, USA. ASME Digital Collection.

[Fowler, 2004] Fowler, M. (2004). **UML distilled: a brief guide to the standard object modeling language**. Addison-Wesley Professional. ISBN 0321193687.

[Frankl and Weyuker, 1988] Frankl, P., and Weyuker, E. (1988). **An Applicable Family of Data Flow Testing Criteria**. *IEEE Transactions on Software Engineering*, 14(10):1483–1498.

[Fritzson, 2010] Fritzson, P. (2010). **Principles of object-oriented modeling and simulation with Modelica 2.1**. John Wiley & Sons. ISBN 978-0-471-47163-9.

[Frost, 1994] Frost, S. (1994). **The Rumbaugh Method (OMT): The Selection of an Object-oriented Analysis Method**. In *Object Development Methods*, pages 189–198. SIGS Publications, Inc.

[Gauthier et al., 2015a] Gauthier, J., Bouquet, F., Hammad, A., and Peureux, F. (2015a). **Tooled Process for Early Validation of SysML Models using Modelica Simulation**. In $Proceedings of the 6^{th}$ *International Conference on Fundamentals of Software Engineering. (FSEN'15)*, LNCS, Tehran, Iran. Springer Berlin Heidelberg.

[Gauthier et al., 2015b] Gauthier, J.-M., Bouquet, F., Hammad, A., and Peureux, F. (2015b). **A SysML Formal Framework to Combine Discrete and Continuous Simulation for Testing**. In *Proceedings of the* 17^{th} *International Conference on Formal Engineering Methods. (ICFEM'15)*, volume * of *LNCS*, pages ***–***, Paris, France. Springer. To appear in the LNCS series.

[Gauthier et al., 2013] Gauthier, J.-M., Bouquet, F., Peureux, F., and Hammad, A. (2013). **Verification and Validation of Meta-Model Based Transformation from SysML to VHDL-AMS**. In *Proceedings of the* 1^{st} *International Conference on Model-Driven Engineering and Software Development. (MODELSWARD'13)*, Barcelona, Spain.

[Goómez et al., 2010] Goómez, C., Pascal, J.-C., Jimenez, J., and Esteban, P. (2010). **Heterogeneous Systems Verification on HiLeS Designer Tool**. In *Proceedings of the 36^{th} Annual Conference on IEEE Industrial Electronics Society. (IECON'10)*, pages 132–137, Glendale, AZ, USA. IEEE Computer Society.

[Gotlieb and Petit, 2006] Gotlieb, A., and Petit, M. (2006). **Path-oriented Random Testing**. In *Proceedings of the 1^{st} International Workshop on Random Testing. (RT'06)*, pages 28–35, Portland, Maine, USA. ACM.

[Griffault et al., 1998] Griffault, A., Lajeunesse, S., Point, G., Rauzy, A., Signoret, J. P., and Thomas, P. (1998). **The Altarica language**. In *Proceedings of the International Conference on Safety and Reliability. (ESREL'98)*, pages 20–24, Trondheim, Norway.

[Grindal et al., 2005] Grindal, M., Offutt, J., and Andler, S. F. (2005). **Combination testing strategies: a survey**. *Software Testing, Verification and Reliability*, 15(3):167–199.

[Grossmann et al., 2011] Grossmann, J., Makedonski, P., Wiesbrock, H.-W., Svacina, J., Schieferdecker, I., and Grabowski, J. (2011). **Model-Based X-in-the-Loop Testing**. In *Model-Based Testing for Embedded Systems*, Series on Computational Analysis, Synthesis, and Design of Dynamic Systems, pages 299–337. CRC Press.

[Group, 2003] Group, O. M. (2003). **MDA Guide Version 1.0.1**. http://www.omg.org/mda/mda_files/MDA_Guide_Version1-0.pdf.

[Halbwachs, 2005] Halbwachs, N. (2005). **A synchronous language at work: the story of Lustre**. In *Proceedings of the 3^{rd} ACM and IEEE International Conference on Formal Methods and Models for Co-Design. (MEMOCODE '05)*, pages 3–11, Verona, Italy. IEEE.

[Harel, 1987] Harel, D. (1987). **Statecharts: A Visual Formalism for Complex Systems**. *Science of Computer Programing*, 8(3):231–274.

[Harel, 2007] Harel, D. (2007). **Statecharts in the Making: A Personal Account**. In *Proceedings of the 3^{rd} ACM SIGPLAN Conference on History of Programming Languages. (HOPL III)*, pages 5–1–5–43, San Diego, California. ACM.

[Hautier and Barre, 2004] Hautier, J.-P., and Barre, P.-J. (2004). **The Causal Ordering Graph A tool for system modelling and control law synthesis**. *Studies in informatics and control*, 13(4):265–284.

[Ho et al., 2000] Ho, P. H., Shiple, T., Harer, K., Kukula, J., Damiano, R., Bertacco, V., Taylor, J., and Long, J. (2000). **Smart simulation using collaborative formal and simulation engines**. In *Proceedings of the 2000 IEEE/ACM International Conference on Computer-aided Design. (ICCAD '00)*, pages 120–126, San Jose, California. IEEE Press.

[Holzmann, 1993] Holzmann, G. (1993). **Design and validation of protocols**. *Computer Network ISDN System*, 25(9):981–1017.

[Holzmann, 1997] Holzmann, G. (1997). **The model checker SPIN**. *IEEE Transactions on Software Engineering*, 23(5):279–295.

[IEEE, 2005] IEEE (2005). **IEEE 1666-2005. IEEE Standard SystemC Language Reference Manual**.

[Iqbal et al., 2011] Iqbal, M., Arcuri, A., and Briand, L. (2011). **Automated System Testing of Real-Time Embedded Systems Based on Environment Models**. Technical Report 2011-19, Simula Research Laboratory.

[Iqbal et al., 2012a] Iqbal, M., Arcuri, A., and Briand, L. (2012a). **Combining Search-Based and Adaptive Random Testing Strategies for Environment Model-Based Testing of Real-Time Embedded Systems**. In *Proceedings of the* 4^{th} *Symposium on Search Based Software Engineering. (SSBSE'12)*, volume 7515 of *LNCS*, pages 136–151, Riva del Garda, Italy. Springer Berlin Heidelberg.

[Iqbal et al., 2015] Iqbal, M., Arcuri, A., and Briand, L. (2015). **Environment modeling and simulation for automated testing of soft real-time embedded software**. *Software and System Modeling*, 14(1):483–524.

[Iqbal et al., 2012b] Iqbal, M. Z., Ali, S., Yue, T., and Briand, L. (2012b). **Experiences of Applying UML/MARTE on Three Industrial Projects**. In *Proceedings of the* 15^{th} *International Conference on Model Driven Engineering Languages and Systems. (MODELS'12)*, volume 7590 of *LNCS*, pages 642–658, Innsbruck, Austria. Springer Berlin Heidelberg.

[ISO, 2005] ISO, O. (2005). **ISO 9000. Quality management systems - Fundamentals and vocabulary**.

[ISO, 2011] ISO, O. (2011). **ISO 26262**. http://www.iso.org/.

[Jacobson et al., 1994] Jacobson, I., Christerson, M., and Constantine, L. L. (1994). **The OOSE Method: A Use-Case-driven Approach**. In *Object Development Methods*, pages 247–270. SIGS Publications, Inc.

[Jard and Jeron, 2005] Jard, C., and Jeron, T. (2005). **TGV: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems**. *International Journal on Software Tools Technology Transfer (STTT)*, 7(4):297–315.

[Jarraya et al., 2007] Jarraya, Y., Soeanu, A., Debbabi, M., and Hassaine, F. (2007). **Automatic Verification and Performance Analysis of Time-Constrained SysML Activity Diagrams**. In *Proceedings of the* 14^{th} *Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems. (ECBS'07)*, pages 515–522, Tucson, Arizona, USA. IEEE Computer Society.

[Ji et al., 2011] Ji, H., Lenord, O., and Schramm, D. (2011). **A Model Driven Approach for Requirements Engineering of Industrial Automation Systems**. In *Proceedings of the* 4^{th} *Workshop on Equation-Based Object-Oriented Modeling Languages and Tools (EOOLT'11)*, pages 9–18, Zürich, Switzerland. Linköping Univ. Electronic Press.

[Johnson et al., 2012] Johnson, T., Kerzhner, A., Paredis, C. J., and Burkhart, R. (2012). **Integrating Models and Simulations of Continuous Dynamics into SysML**. *Journal of Computing and Information Science in Engineering*, 12(1):13–24.

[Kawahara et al., 2009] Kawahara, R., Nakamura, H., Dotan, D., Kirshin, A., Sakairi, T., Hirose, S., Ono, K., and Ishikawa, H. (2009). **Verification of Embedded System's Specification Using Collaborative Simulation of SysML and Simulink models**. In *Proceedings of the 4^{th} International Conference on Model-Based Systems Engineering. (MBSE'09)*, pages 21–28, Haifa, Israël. IEEE Computer Society.

[Kleijnen, 1995] Kleijnen, J. P. (1995). **Verification and Validation of Simulation Models**. *European Journal of Operational Research*, 82(1):145–162.

[Knorreck et al., 2011] Knorreck, D., Apvrille, L., and de Saqui-Sannes, P. (2011). **TEPE: A SysML Language for Time-constrained Property Modeling and Formal Verification**. *SIGSOFT Software Engineering Notes*, 36(1):1–8.

[Krichen and Tripakis, 2009] Krichen, M., and Tripakis, S. (2009). **Conformance Testing for Real-Time Systems**. *Formal Methods in System Design*, 34(3):238–304.

[Kripke, 1959] Kripke, S. A. (1959). **A completeness theorem in modal logic**. *Journal of Symbolic Logic*, 24(1):1–14.

[Kuchcinski and Szymanek, 2013] Kuchcinski, K., and Szymanek, R. (2013). **Jacop-java constraint programming solver**. In *Workshop CP Solvers: Modeling, Applications, Integration, and Standardization, held at the 19^{th} International Conference on Principles and Practice of Constraint Programming. (CP'13)*, volume 5195 of *Lecture Notes in Computer Science*, Uppsala, Sweden. Springer Berlin Heidelberg.

[Larsen et al., 1997] Larsen, K. G., Pettersson, P., and Yi, W. (1997). **Uppaal in a Nutshell**. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152.

[Lasalle et al., 2011a] Lasalle, J., Peureux, F., and Fondement, F. (2011a). **Development of an automated MBT toolchain from UML/SysML models**. *Innovations in Systems and Software Engineering*, 7(4):247–256.

[Lasalle et al., 2011b] Lasalle, J., Peureux, F., and J., G. (2011b). **Automatic test concretization to supply end-to-end MBT for automotive mechatronic systems**. In *Proceedings of the 1^{st} International Workshop on End-to-End Test Script Engineering. (ETSE '11)*, pages 16—23, Toronto, Canada. ACM.

[Le Lann, 1998] Le Lann, G. (1998). **Proof-Based System Engineering and Embedded Systems**. In *Lectures on Embedded Systems, European Educational Forum, School on Embedded Systems*, volume 1494 of *LNCS*, pages 208–248, Veldhoven, Netherlands. Springer Berlin Heidelberg.

[Leavens et al., 2006] Leavens, G., Baker, A., and Ruby, C. (2006). **Preliminary design of JML: a behavioral interface specification language for JAVA**. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38.

[Leavens et al., 1998] Leavens, G. T., Baker, A. L., and Ruby, C. (1998). **JML: a Java modeling language**. In *Formal Underpinnings of Java Workshop. (OOPSLA'98)*, pages 404–420.

[Legeard and Legros, 1991] Legeard, B., and Legros, E. (1991). **Short overview of the CLPS system**. In *Proceeding of the 3^{rd} International Symposium on Programming Language Implementation and Logic Programming. (PLILP'91)*, volume 528 of *LNCS*, pages 431–433. Springer Berlin Heidelberg, Passau, Germany.

[Legeard et al., 2002] Legeard, B., Peureux, F., and Utting, M. (2002). **Automated Boundary Testing from Z and B**. In *Proceedings of the International Symposium of Formal Methods Europe. (FME'02)*, volume 2391 of *LNCS*, pages 21–40, Copenhagen, Denmark. Springer Berlin Heidelberg.

[Legeard et al., 2004] Legeard, B., Peureux, F., and Utting, M. (2004). **Controlling test case explosion in test generation from B formal models**. *Software Testing, Verification and Reliability*, 14(2):81–103.

[Legeard and Py, 1999] Legeard, B., and Py, L. (1999). **Constraint Logic Programming with Sets for animation and verification of B specification**. In *Proceeding of the International Workshop on Declarative Programming Systems. (DPS'99)*, Paris, France.

[Lettrari and Klose, 2001] Lettrari, M., and Klose, J. (2001). **Scenario-based monitoring and testing of real-time uml models**. In *Proceedings of the* 4^{th} *International Conference on The Unified Modeling Language. Modeling Languages, Concepts, and Tools. (UML'01)*, volume 2185 of *LNCS*, pages 317–328, Toronto, Canada. Springer Berlin Heidelberg.

[Li et al., 2004] Li, X., Liu, Z., and Jifeng, H. (2004). **A formal semantics of UML sequence diagram**. In *Proceedings of the Australian Software Engineering Conference. (ASWEC'04)*, pages 168–177, Melbourne, Australia. IEEE.

[Locment and Sechilariu, 2010] Locment, F., and Sechilariu, M. (2010). **Energetic Macroscopic Representation and Maximum Control Structure of electric vehicles charging photovoltaic system**. In *Proceedings of the IEEE Vehicle Power and Propulsion Conference. (VPPC'10)*, pages 1–6, Lille, France. IEEE.

[Logrippo et al., 1992] Logrippo, L., Faci, M., and Haj-Hussein, M. (1992). **An introduction to LOTOS: learning by examples**. *Computer Networks and ISDN Systems*, 23(5):325–342.

[Lundvall and Fritzson, 2005] Lundvall, H., and Fritzson, P. (2005). **Event Handling in the OpenModelica Compiler and Run-time System**. In *Proceedings of the* 46^{th} *Conference on Simulation and Modelling of the Scandinavian Simulation Society. (SIMS'05)*, Trondheim, Norway. Linkoping University Electronic Press.

[Macworth, 1977] Macworth, A. K. (1977). **Consistency in networks of relations**. *Journal of Artificial Intelligence*, 8(1):99–118.

[Matignon et al., 2010] Matignon, L., Laurent, G. J., Le Fort-Piat, N., and Chapuis, Y.-A. (2010). **Designing decentralized controllers for distributed-air-jet mems-based micromanipulators by reinforcement learning**. *Journal of intelligent & robotic systems*, 59(2):145–166.

[Matinnejad et al., 2014] Matinnejad, R., Nejati, S., Briand, L., and Bruckmann, T. (2014). **MiL Testing of Highly Configurable Continuous Controllers: Scalable Search Using Surrogate Models**. In 29^{th} *ACM/IEEE International Conference on Automated Software Engineering (ASE'14)*, pages 163–174, Vasteras, Sweden. ACM.

[Matinnejad et al., 2013] Matinnejad, R., Nejati, S., Briand, L., Bruckmann, T., and Poull, C. (2013). **Automated Model-in-the-Loop Testing of Continuous Controllers Using Search**. In *Proceedings of the* 5^{th} *Symposium on Search Based Software Engineering.*

(SSBSE'13), volume 8084 of *LNCS*, pages 141–157. Springer Berlin Heidelberg, St Petersburg, Russia.

[McMinn, 2011] McMinn, P. (2011). **Search-Based Software Testing: Past, Present and Future**. In *Proceedings of the IEEE* 4^{th} *International Conference on Software Testing, Verification and Validation Workshops. (ICSTW'11)*, pages 153–163, Berlin, Germany. IEEE.

[McUmber and Cheng, 1999] McUmber, W. E., and Cheng, B. H. C. (1999). **UML-Based Analysis of Embedded Systems Using a Mapping to VHDL**. In *Proceedings of the* 4^{th} *IEEE International Symposium on High-Assurance Systems Engineering. (HASE'99)*, pages 56–63, Washington, DC, USA. IEEE Computer Society.

[Meyer, 1988] Meyer, B. (1988). **Eiffel: A language and environment for software engineering**. *Journal of Systems and Software*, 8(3):199–246.

[Mikucionis et al., 2003] Mikucionis, M., Larsen, K., and Nielsen, B. (2003). **Online on-the-fly testing of real-time systems**. Technical Report RS-03-49, Basic Research In Computer Science (BRICS).

[Mikucionis et al., 2004] Mikucionis, M., Larsen, K., and Nielsen, B. (2004). **T-UPPAAL: online model-based testing of real-time systems**. In *Proceedings of the* 19^{th} *International Conference on Automated Software Engineering. (ASE'04)*, pages 396–397, Linz, Austria. IEEE.

[Milner, 1999] Milner, R. (1999). **Communicating and Mobile Processes: the π-calculus**. Cambridge University Press. ISBN 0-5216-5869-1.

[Myers et al., 2011] Myers, G. J., Sandler, C., and Badgett, T. (2011). **The Art of Software Testing**. Wiley Publishing, 3^{rd} edition. ISBN 9781118031964.

[Nytsch-Geusen, 2007] Nytsch-Geusen, C. (2007). **The use of the UML within the modeling process of Modelica-models.** In *Proceedings of the* 1^{st} *Workshop on Equation-Based Object-Oriented Modeling Languages and Tools (EOOLT'07)*, pages 1–11, Berlin, Germany. Linköping Univ. Electronic Press.

[Offut et al., 1999] Offut, A., Xiong, Y., and Liu, S. (1999). **Criteria for generating specification-based tests**. In *Proceedings of the* 5^{th} *International Conference on Engineering of Complex Computer Systems. (ICECCS'99)*, pages 119–131, Las-Vegas, USA. IEEE Computer Society Press.

[OMG, 2008] OMG (2008). **MOF Model to Text Transformation Language (MOFM2T), 1.0**. http://www.omg.org/spec/MOFM2T/1.0/.

[OMG, 2011] OMG (2011). **UML profile for MARTE**. http://www.omg.org/spec/MARTE/.

[OMG, 2012] OMG (2012). **Systems Modeling Language (SysML), Version 1.3**. http://www.omg.org/SysML/1.3/.

[OMGSM, 2012] OMGSM (2012). **OMG SysML-Modelica Transformation specification V1.0**. http://www.omg.org/spec/SyM/1.0/.

[Pedroza et al., 2011] Pedroza, G., Apvrille, L., and Knorreck, D. (2011). **AVATAR: A SysML Environment for the Formal Verification of Safety and Security Properties**. In *Proceedings of the* 11^{th} *International Conference on New Technologies of Distributed Systems. (NOTERE'11)*, pages 1–10, Paris, France. IEEE Computer Society.

[Peñil et al., 2010] Peñil, P., Medina, J., Posadas, H., and Villar, E. (2010). **Generating Heterogeneous Executable Specifications in SystemC from UML/MARTE Models**. *Innovations in Systems and Software Engineering*, 6(1-2):65–71.

[Peterson et al., 2002] Peterson, G., Ashenden, P., and Teegarden, D. (2002). **The System Designer's Guide to VHDL-AMS**. Morgan Kaufmann Publishers Inc. ISBN 1558607498.

[Pop et al., 2007] Pop, A., Akhvlediani, D., and Fritzson, P. (2007). **Towards unified system modeling with the modelicaml UML profile**. In *Proceedings of the* 1^{st} *Workshop on Equation-Based Object-Oriented Modeling Languages and Tools (EOOLT'07)*, pages 13–24, Berlin, Germany. Linköping Univ. Electronic Press.

[Pretschner, 2005] Pretschner, A. (2005). **Model-Based Testing**. In *Proceedings of the* 27^{th} *International Conference on Software Engineering. (ICSE'05)*, pages 722–723, St. Louis, USA. ACM.

[Qamar et al., 2009] Qamar, A., During, C., and Wikander, J. (2009). **Designing Mechatronic Systems, a Model-Based Perspective, an Attempt to Achieve SysML-Matlab/Simulink Model Integration**. In *Proceedings of the International Conference on Advanced Intelligent Mechatronics. (AIM'09)*, pages 1306–1311, Singapore, Republic of Singapore. IEEE Computer Society.

[Rashid et al., 2015a] Rashid, M., Anwar, M. W., and M., K. A. (2015a). **A Systematic Investigation of Tools in Model Based System Engineering for Embedded Systems**. In *Proceedings of the* 10^{th} *IEEE International Conference on System of Systems Engineering. (SoSE'15)*, page to appear, San Antonio, USA. IEEE.

[Rashid et al., 2015b] Rashid, M., Anwar, M. W., and M., K. A. (2015b). **Identification of Trends for Model Based Development of Embedded Systems**. In *Proceedings of the* 12^{th} *IEEE International Symposium on Programming and Systems. (ISPS'15)*, page to appear, Algiers, Algeria. IEEE.

[Raslan and Sameh, 2007] Raslan, W., and Sameh, A. (2007). **System-level modeling and design using SysML and SystemC**. In *Proceedings of the International Symposium on Integrated Circuits (ISIC'07)*, pages 504–507, Singapore, Republic of Singapore. IEEE Computer Society.

[Reichwein et al., 2012] Reichwein, A., Paredis, C. J., Canedo, A., Witschel, P., Stelzig, P. E., Votintseva, A., and Wasgint, R. (2012). **Maintaining consistency between system architecture and dynamic system models with SysML4Modelica**. In *Proceedings of the* 6^{th} *International Workshop on Multi-Paradigm Modeling. (MPM'12)*, pages 43–48, Innsbruck, Austria. ACM.

[Rieder et al., 2006] Rieder, M., Steiner, R., Berthouzoz, C., Corthay, F., and Sterren, T. (2006). **Synthesized UML, a Practical Approach to Map UML to VHDL**. In *Proceedings of the* 2^{nd} *International Conference on Rapid Integration of Software Engineering Techniques. (RISE'05)*, volume 3943 of *LNCS*, pages 203–217, Heraklion, Crete, Greece. Springer Berlin Heidelberg.

[Schamai et al., 2009] Schamai, W., Fritzson, P., Paredis, C., and Pop, A. (2009). **Towards unified system modeling and simulation with ModelicaML: modeling of executable behavior using graphical notations**. In *Proceedings of the* 7^{th} *Modelica Conference*, pages 612–621, Como, Italy. Linköping Univ. Electronic Press.

[Shirole and Kumar, 2013] Shirole, M., and Kumar, R. (2013). **UML Behavioral Model Based Test Case Generation: A Survey**. *SIGSOFT Software Engineering Notes*, 38(4):1–13.

[Sindico et al., 2011] Sindico, A., Di Natale, M., and Panci, G. (2011). **Integrating SysML with Simulink using Open-Source Model Transformations**. In *Proceedings of the* 1^{st} *International Conference on Simulation and Modeling Methodologies, Technologies and Applications. (SIMULTECH'11)*, pages 45–56, Noordwijkerhout, The Netherlands. SciTePress Digital Library.

[Sjöstedt et al., 2007] Sjöstedt, C.-J., Shi, J., Törngren, M., Servat, D., Chen, D., Ahlsten, V., and Lönn, H. (2007). **Mapping Simulink to UML in the Design of Embedded Systems: Investigating Scenarios and Transformations**. In *Proceedings of the* 4^{th} *Workshop on Object-oriented Modeling of Embedded Real-Time Systems. (OMER'07)*, pages 137–160, Paderborn, Germany. Citeseer.

[Spivey, 1992a] Spivey, J. (1992a). **The Z Notation**. Prentice Hall, 2^{nd} edition. ISBN 2-225-84367-8.

[Spivey, 1992b] Spivey, J. (1992b). **The Z notation: A Reference Manual**. Prentice-Hall, 2^{nd} edition. ISBN 0-1397-8529-9.

[Sueur and Dauphin-Tanguy, 1991] Sueur, C., and Dauphin-Tanguy, G. (1991). **Bond-graph approach for structural analysis of MIMO linear systems**. *Journal of the Franklin Institute*, 328(1):55–70.

[Sutton et al., 2007] Sutton, M., Greene, A., and Amini, P. (2007). **Fuzzing: Brute Force Vulnerability Discovery**. Addison-Wesley Professional.

[Utting and Legeard, 2007] Utting, M., and Legeard, B. (2007). **Practical Model-Based Testing: A Tools Approach**. Morgan Kaufmann Publishers Inc. ISBN 9780080466484.

[Utting et al., 2012] Utting, M., Pretschner, A., and Legeard, B. (2012). **A taxonomy of model-based testing approaches**. *Software Testing, Verification and Reliability*, 22(5):297–312.

[Vanderperren and Dehaene, 2006] Vanderperren, Y., and Dehaene, W. (2006). **From UML/SysML to Matlab/Simulink: Current State and Future Perspectives**. In *Proceedings of the* 9^{th} *Conference on Design, Automation and Test in Europe. (DATE'06)*, pages 93–93, Munich, Germany. European Design and Automation Association.

[Vangheluwe et al., 2002] Vangheluwe, H., De Lara, J., and Mosterman, P. J. (2002). **An introduction to multi-paradigm modelling and simulation**. In *Proceedings of the Artificial Intelligence, Simulation and Planning in High Autonomy Systems. (AIS'02)*, pages 9–20, Lisboa, Potugal. SCS.

[Vasaiely, 2009] Vasaiely, P. (2009). **Interactive simulation of SysML models using Modelica**. *Hamburg University of Applied Sciences*.

[Vouk, 1990] Vouk, M. (1990). **Back-to-Back Testing**. *Information and Software Technology*, 32(1):34–45.

[Warmer and Kleppe, 1996] Warmer, J., and Kleppe, A. (1996). **The Object Constraint Language: Precise Modeling with UML**. Addison-Wesley. ISBN 0-2013-7940-6.

[White, 1994] White, I. (1994). **Rational Rose Essentials: Using the Booch Method**. Addison-Wesley Longman Publishing Co., Inc., 1^{st} edition. ISBN 0805306161.

[Wilhelm et al., 2008] Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., and Stenström, P. (2008). **The Worst-case Execution-time Problem&Mdash;Overview of Methods and Survey of Tools**. *ACM Transactions on Embedded Computing Systems*, 7(3):36:1–36:53.

[Wood et al., 2008] Wood, S., Akehurst, D., Uzenkov, O., Howells, W., and McDonald-Maier, K. (2008). **A Model-Driven Development Approach to Mapping UML State Diagrams to Synthesizable VHDL**. *IEEE Transactions on Computers*, 57(10):1357–1371.

[Wymore, 1993] Wymore, A. W. (1993). **Model-Based Systems Engineering**. CRC Press, Inc., 1^{st} edition. ISBN 084938012X.

[Yi et al., 1994] Yi, W., Pettersson, P., and Daniels, M. (1994). **Automatic Verification of Real-Time Communicating Systems By Constraint-Solving**. In *Proceedings of the 7^{th} International Conference on Formal Description Techniques. (FORTE'94)*, pages 223–238, Berne, Switzerland. North–Holland.

[Zeigler et al., 2000] Zeigler, B. P., Kim, T. G., and Praehofer, H. (2000). **Theory of Modeling and Simulation**. Academic Press, Inc., Orlando, FL, USA, 2^{nd} edition. ISBN: 0127784551.

V

APPENDIX

A

LANGUAGE SUBSETS

```
[ ] optional
{ } repeat zero or more times

constraint:
 {equation ";"}

equation :
 simple_expression "=" expression
 | if_equation
 | for_equation
 | connect_clause
 | when_equation
 | IDENT function_call_args )

if_equation :
 if expression then
  { equation ";" }
 { elseif expression then
  { equation ";" }
 }
 [ else
  { equation ";" }
 ]
 end if

for_equation :
 for for_indices loop
  { equation ";" }
 end for

when_equation :
 when expression then
  { equation ";" }
 { elsewhen expression then
  { equation ";" }
 }
 end when

connect_clause :
 connect "(" component_reference "," component_reference ")"

component_reference :
 [ "." ] IDENT { "." IDENT }
```

Figure A.1: Modelica Subset for Equation Specification

```
[ ] optional
{ } repeat zero or more times

guard:
 expression

expression:
 simple_expression

simple_expression:
 logical_expression

logical_expression :
 logical_term { or logical_term }

logical_term :
 logical_factor { and logical_factor }

logical_factor :
 [ not ] relation

relation :
 arithmetic_expression rel_op arithmetic_expression

rel_op :
 "<" | "<=" | ">" | ">=" | "==" | "<>"

arithmetic_expression :
 [ add_op ] term { add_op term }

add_op :
 "+" | "-" | ".+" | ".-"

term :
 factor { mul_op factor }

mul_op :
 "*" | "/" | ".*" | "./"

factor :
 primary [ ("^" | ".^") primary ]

primary :
 UNSIGNED_NUMBER
 | STRING
 | false
 | true
 | component_reference

component_reference :
 [ "." ] IDENT { "." IDENT }
```

Figure A.2: Modelica Subset for Guard Specification

```
[ ] optional
{ } repeat zero or more times

effect:
 {statement ";"}

statement :
 component_reference ( ":=" expression | function_call_args )
 | "(" output_expression_list ")" ":=" component_reference function_call_args
 | break
 | return
 | if_statement
 | for_statement
 | while_statement
 | when_statement

if_statement :
 if expression then
  { statement ";" }
 { elseif expression then
  { statement ";" }
 }
 [ else
  { statement ";" }
 ]
 end if

for_statement :
 for for_indices loop
  { statement ";" }
 end for

for_indices :
 for_index {"," for_index}

for_index:
 IDENT [ in expression ]

while_statement :
 while expression loop
  { statement ";" }
 end while

when_statement :
 when expression then
  { statement ";" }
 { elsewhen expression then
  { statement ";" }
 }
 end when
```

Figure A.3: Modelica Subset for Effect Specification

```
[ ] optional
{ } repeat zero or more times

precondition :
 oclExpression

oclExpression :
 logicalExpression

logicalExpression :
 relationalExpression {
  and relationalExpression
 | or relationalExpression
 | xor relationalExpression
 }

relationalExpression :
 additiveExpression {
  '<' additiveExpression
  | '>' additiveExpression
  | '<=' additiveExpression
  | '>=' additiveExpression
  | '<>' additiveExpression
  | '=' additiveExpression
 }

additiveExpression :
 multiplicativeExpression {
  '+' multiplicativeExpression
  | '-' multiplicativeExpression
 }

multiplicativeExpression :
 unaryExpression {
 '*' unaryExpression
  | 'div' unaryExpression
  | 'mod' unaryExpression
 }

unaryExpression :
 ['-' | 'not'] postfixExpression

postfixExpression :
 primaryExpression {
  dotSuffix
  | arrowSuffix
 }

dotSuffix :
 : "." ocl_call
 | "." link

ocl_call :
 ocl_operation "("[ocl_parameter {"," ocl_parameter}]")"

link :
 IDENT

ocl_operation :
 "allInstances"
 | "oclIsUndefined"
 | IDENT

ocl_parameters :
 "(" [ocl_parameter {"," ocl_parameter}] ")"

ocl_parameter :
 postfixExpression
```

Figure A.4: OCL Subset for SysML Models Part One

```
ifExpression :
 if logicalExpression then
  logicalExpression
 else
  logicalExpression
 endif

letExpression :
 let letDefinition in logicalExpression

letDefinitionList :
 letDefinition {"," letDefinition}

letDefinition :
 IDENT "=" logicalExpression
 | IDENT "," type "=" logicalExpression

arrowSuffix :
 "->" setOperator

setOperator :
 IDENT "()"
 | IDENT "(" IDENT "," type "|" logicalExpression ")"
 | IDENT "(" logicalExpression ")"

type :
 IDENT
 | "range( Integer, " INT ".." INT ")"
 | "Integer"
 | IDENT "(" IDENT ")"
 | "Boolean"


primaryExpression :
 "(" logicalExpression ")"
 | IDENT
 | "self"
 | INT
 | enumLiteral
 | TRUE
 | FALSE
 | ifExpression
 | letExpression
 | ocl_call

enumLiteral :
 IDENT "::" IDENTIFIER
 | IDENT "::" var

var :
 "$" IDENT
```

Figure A.5: OCL Subset for SysML Models Part Two

B

MODELS AND GENERATED CODE

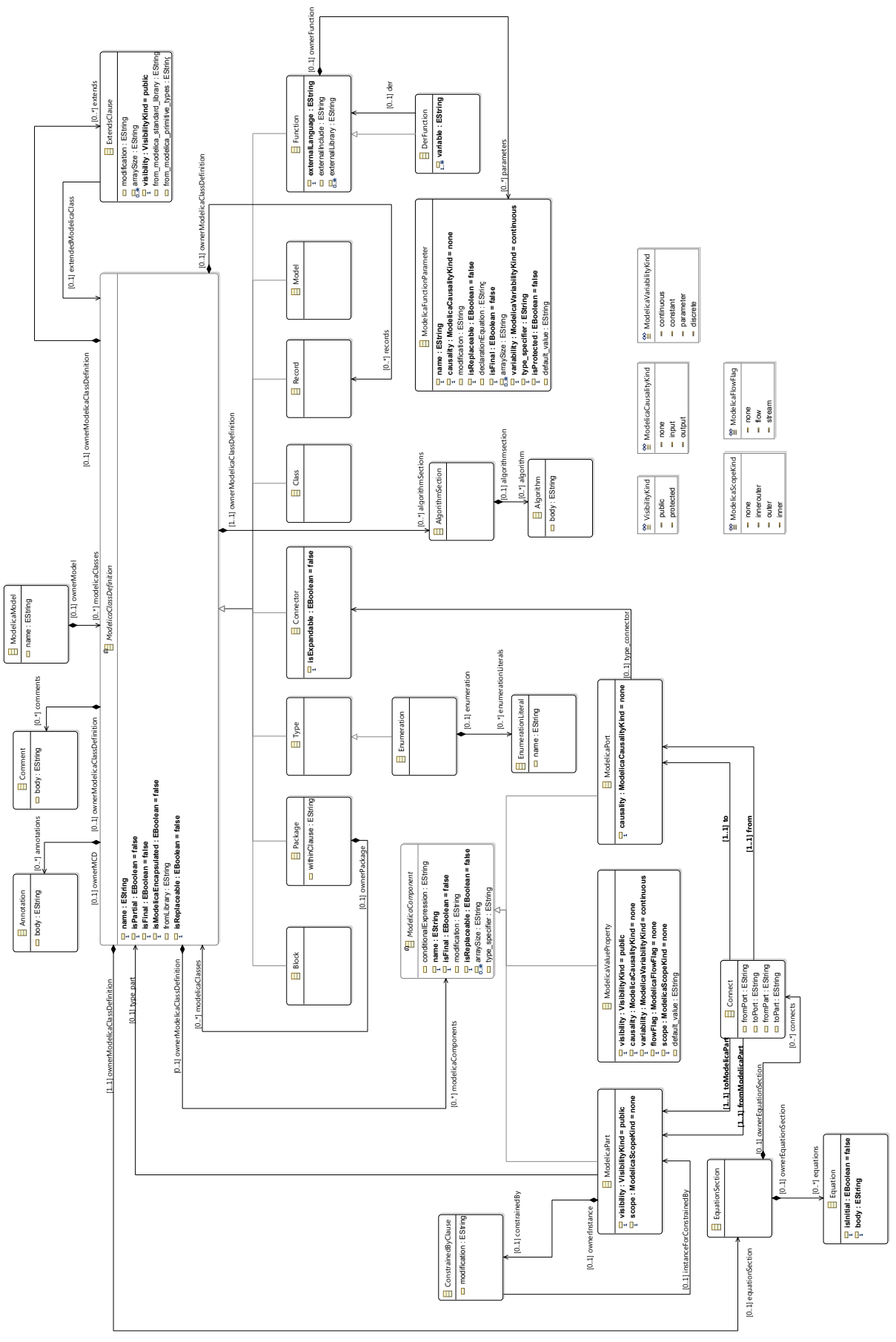


Figure B.1: Proposed Modelica Meta-model

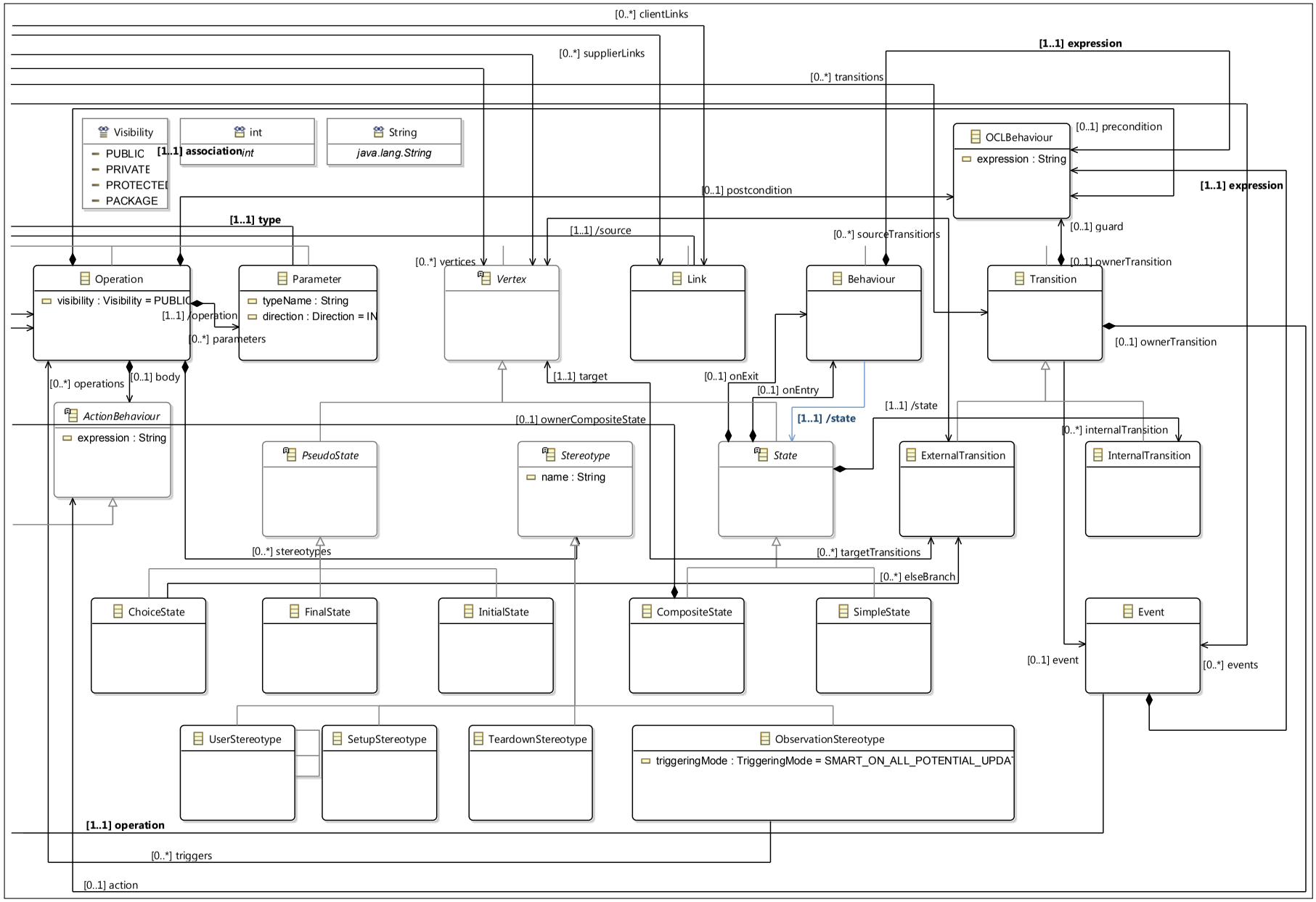


Figure B.2: UML4TST Meta-model Part One

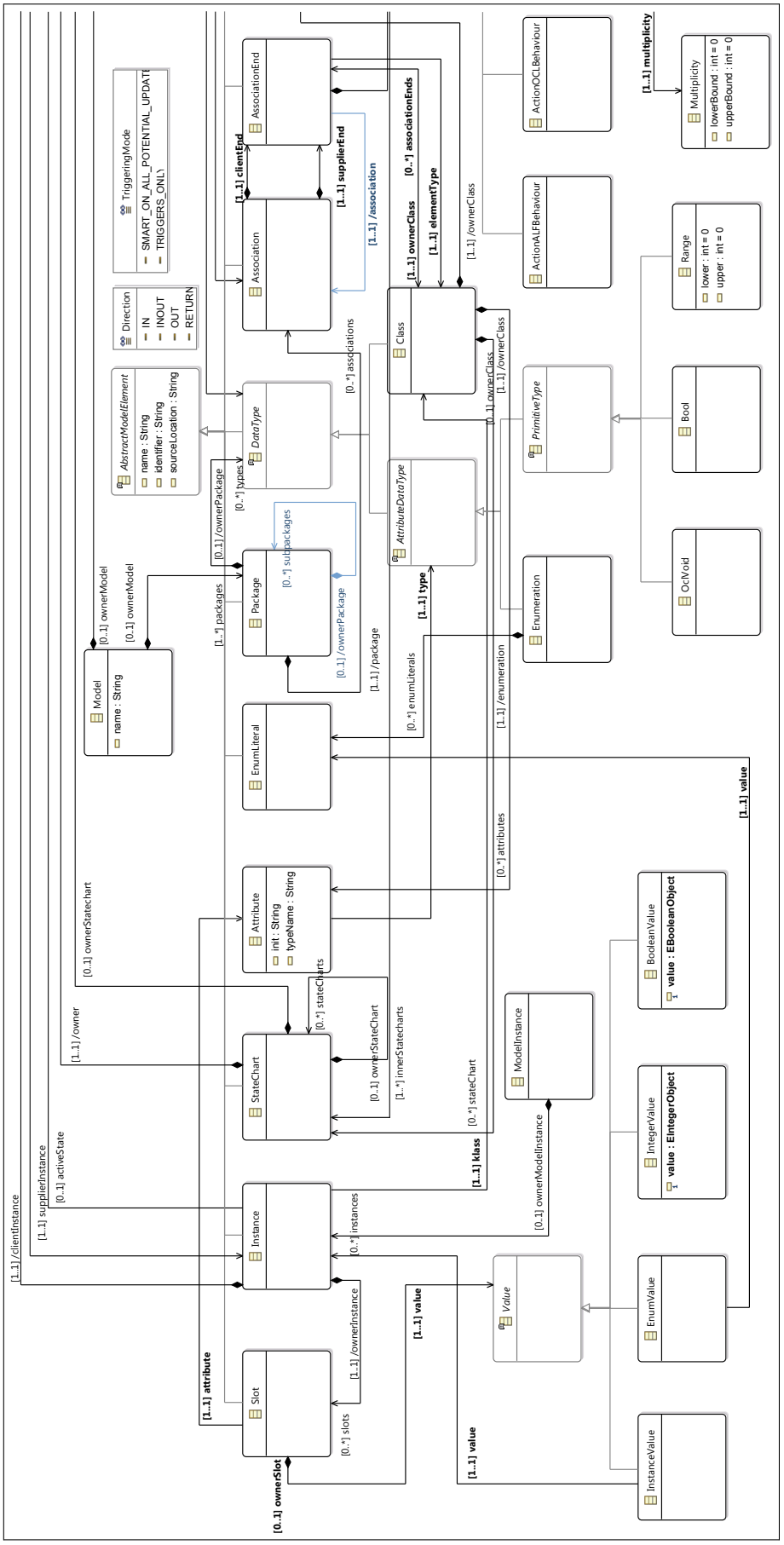


Figure B.3: UML4TST Meta-model Part Two

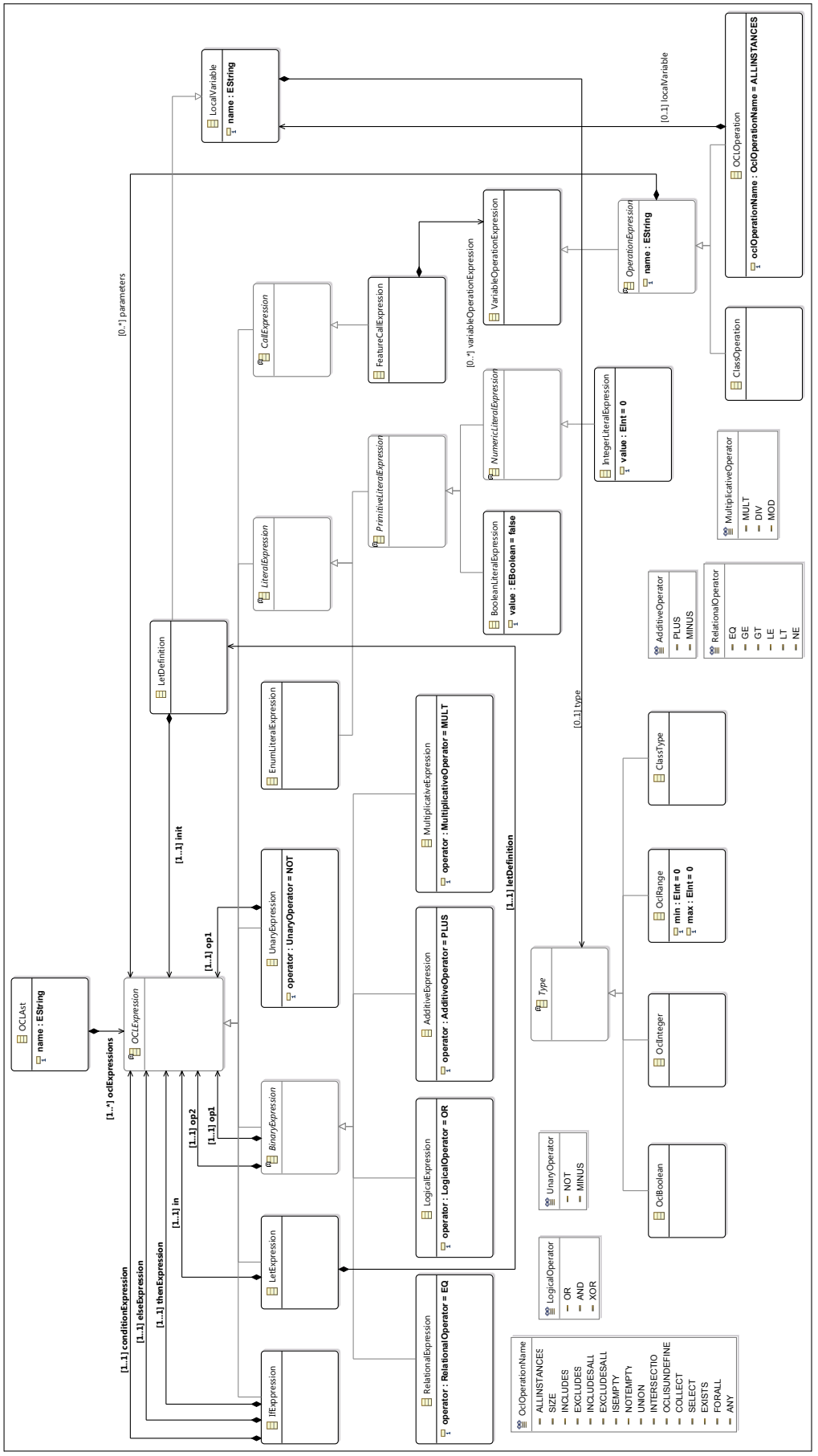


Figure B.4: OCL4TST Meta-model

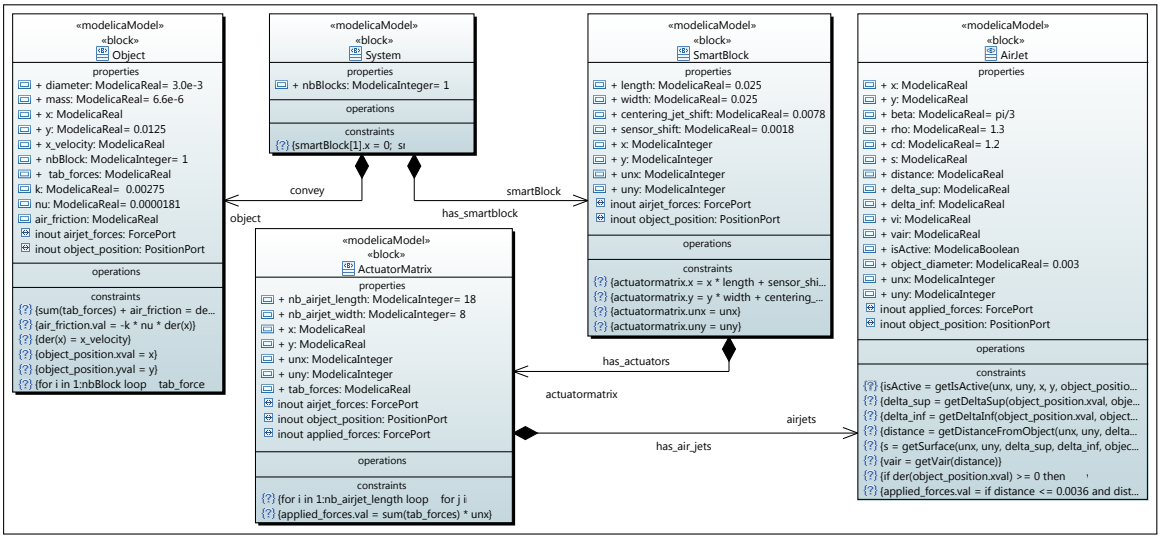


Figure B.5: The SmartBlock BDD

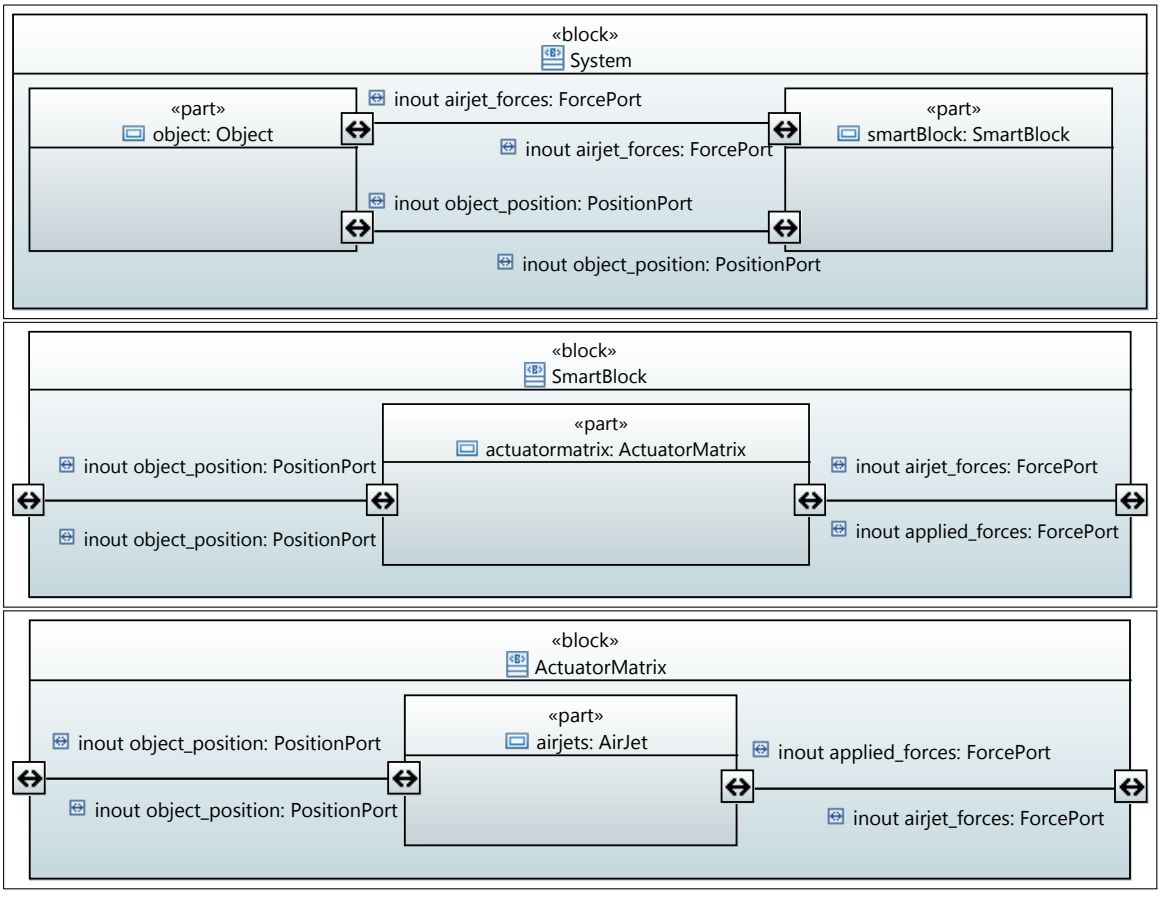


Figure B.6: The SmartBlocks System IBDs

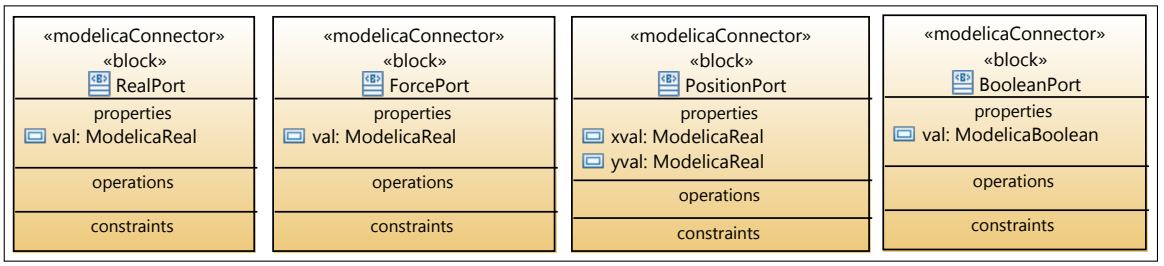


Figure B.7: Modelica Connectors for Flow Ports Typing

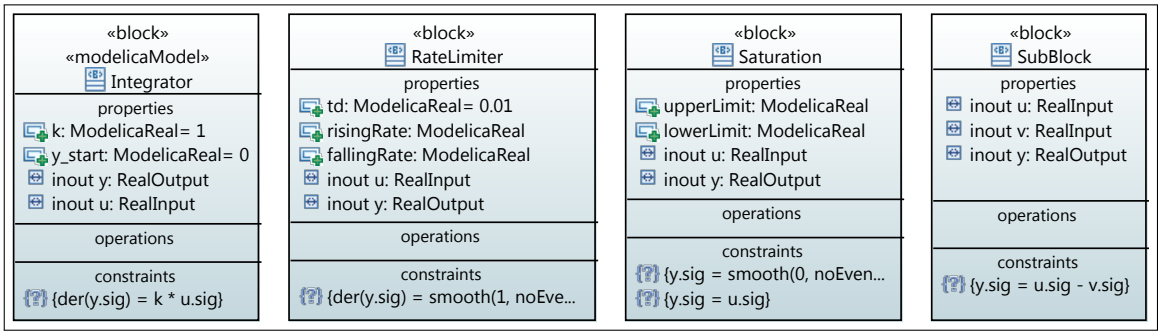


Figure B.8: SysML Block Definition Diagram for Mathematical Blocks

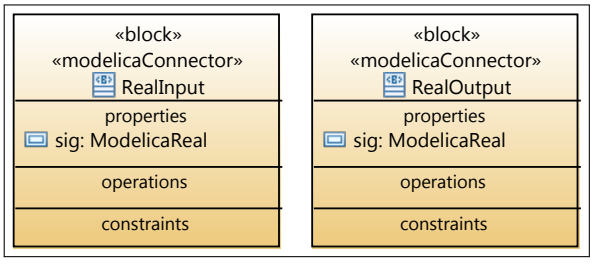


Figure B.9: SysML Block Definition Diagram of Connectors

```
model Tank
  /* Generated with the plugin UFC - SysML4Modelica
  */
  parameter Real area(unit = "m2") = 1.0;
  parameter Real flowGain(unit = "m2/s") = 0.05;
  parameter Real minV = 0.0;
  parameter Real maxV = 10.0;
  Real h(unit = "m", start = 0.0);
  Two_tanks.Design.Interface.LiquidFlow fIn;
  Two_tanks.Design.Interface.ActSignal tActuator;
  Two_tanks.Design.Interface.ReadSignal tSensor;
  Two_tanks.Design.Interface.LiquidFlow fOut;
  Boolean Initial1;
  Boolean Empty;
  Boolean PartiallyFilled;
  Boolean OverFlow;
equation
  assert(minV >= 0, "minV - minimum Valve level must be >= 0");
  der(h) = (fIn.val - fOut.val) / area;
  fOut.val = limitValue(minV, maxV, -flowGain * tActuator.val);
  tSensor.val = h;
algorithm
  when initial() then
    Initial1 := true;
  end when;
  when pre(Initial1) then
    Initial1 := false;
    Empty := true;
  end when;
  when pre(Empty) and h > 0.2 then
    Empty := false;
    PartiallyFilled := true;
  end when;
  when pre(PartiallyFilled) and h <= 0.2 then
    PartiallyFilled := false;
    Empty := true;
  end when;
  when pre(PartiallyFilled) and h > 0.8 then
    PartiallyFilled := false;
    OverFlow := true;
  end when;
  when pre(OverFlow) and h <= 0.8 then
    OverFlow := false;
    PartiallyFilled := true;
  end when;
end Tank;
```

Figure B.10: Generated Modelica Code from the Tank Block

```
model Environment
 Two_tanks.Design.Interface.LiquidFlow sOut;
 Two_tanks.Design.Components.LiquidSource liquidSource;
 Boolean activateLowFlowLevelEvent;
 Boolean activateHighFlowLevelEvent;
 Boolean stopFlowLevelEvent;
 Boolean initEnvEvent;
 Boolean Initial1;
 Boolean LowFlowState;
 Boolean HighFlowState;
 Boolean NoFlowState;

equation
 connect(liquidSource.sOut, sOut);

algorithm
 when initial() then
  Initial1 := true;
  //Initial1.onEntry
 end when;

 when pre(Initial1) [and transition guard] [and op precondition] then
  //op postcondition
  //Initial1.onExit
  liquidSource.sourceValue := 0.0; //transition.effect
  //NoFlowState.onEntry
  Initial1:= false;
  NoFlowState:= true;
 end when;

 when pre(NoFlowState) and activateLowFlowLevelEvent [and transition guard] [and op
     precondition] then
  //op postcondition
  //NoFlowState.onExit
  liquidSource.sourceValue := 0.02;//transition.effect
  //LowFlowState.onEntry
  NoFlowState:= false;
  LowFlowState:= true;
 end when;

 when pre(LowFlowState) and activateHighFlowLevelEvent [and transition guard] [and op
     precondition] then
  //op postcondition
  //LowFlowState.onExit
  liquidSource.sourceValue := 0.06; //transition.effect
  //HighFlowState.onEntry
  LowFlowState:= false;
  HighFlowState:= true;
 end when;

 when pre(HighFlowState) and activateLowFlowLevelEvent [and transition guard] [and op
     precondition] then
  //op postcondition
  //HighFlowState.onExit
  liquidSource.sourceValue := 0.02; //transition.effect
  //LowFlowState.onEntry
  HighFlowState:= false;
  LowFlowState:= true;
 end when;

 when pre(LowFlowState) and stopFlowLevelEvent [and transition guard] [and op
     precondition] then
  //op postcondition
  //LowFlowState.onExit
  liquidSource.sourceValue := 0.0; //transition.effect
  //NoFlowState.onEntry
  LowFlowState:= false;
  NoFlowState:= true;
 end when;
end Environment;
```

Figure B.11: Generated Modelica Code of the Environment Block

```
model TestCase1
/* Generated with the plugin UFC - SysML4Modelica
*/
 Two_tanks.Design.Components.TankSystem system;

algorithm
when initial() then
 system.environment.stopFlowLevelEvent := false;
 system.environment.activateHighFlowLevelEvent := false;
 system.environment.activateLowFlowLevelEvent := false;
 end when;
 when time > 0 and time <= 150 then
 system.environment.activateLowFlowLevelEvent := true;
 system.environment.stopFlowLevelEvent := false;
 system.environment.activateHighFlowLevelEvent := false;
 end when;
 when time > 150 and time <= 350 then
 system.environment.activateHighFlowLevelEvent := true;
 system.environment.stopFlowLevelEvent := false;
 system.environment.activateLowFlowLevelEvent := false;
 end when;
 when time > 350 and time <= 600 then
 system.environment.activateLowFlowLevelEvent := true;
 system.environment.stopFlowLevelEvent := false;
 system.environment.activateHighFlowLevelEvent := false;
 end when;
 when time > 600 and time <= 650 then
 system.environment.stopFlowLevelEvent := true;
 system.environment.activateHighFlowLevelEvent := false;
 system.environment.activateLowFlowLevelEvent := false;
 end when;
 when time > 650 then
 system.environment.stopFlowLevelEvent := false;
 system.environment.activateHighFlowLevelEvent := false;
 system.environment.activateLowFlowLevelEvent := false;
 end when;
end TestCase1;
```

Figure B.12: Concretization into Modelica Code of the Test Sequence

```
context(c_Environment).

%Declaration of the set all_instances of the class Environment
declaration(id_13, c_Environment, static, s(all_instances), set(atom), {none,
    ins_environment}).
predicat(c_Environment, static, id_20, c_Environment dot s(all_instances) =  {none,
    ins_environment}).

%Declaration of the variable instances of the class Environment
declaration(id_14, c_Environment, variable, v(instances), set(atom), {none,
    ins_environment}).
predicat(c_Environment, invariant, id_21, c_Environment dot v(instances) <: c_Environment
     dot s(all_instances) minus {none}).

%Declaration of the variable currentInstance of the class Environment
declaration(id_15, c_Environment, variable, v(currentInstance), atom, {none,
    ins_environment}).
predicat(c_Environment, invariant, id_22, c_Environment dot v(currentInstance) :
    c_Environment dot v(instances) \/ {none}).

%Initialisation of the set of instances of the class Environment
predicat(c_Environment, initialisation, id_23, c_Environment dot v(instances) = {
    ins_environment}).

%Initialisation of the current instance of the class Environment
predicat(c_Environment, initialisation, id_24, c_Environment dot v(currentInstance) =
    ins_environment).

%Constructor for the class Environment
createInstance(m_Two_tanks, c_Environment, construct_c_Environment).
operation(id_16, m_Two_tanks, construct_c_Environment).
declaration(id_17, m_Two_tanks, input(construct_c_Environment), i(arg_instance), atom, {
    none, ins_environment}).
predicat(m_Two_tanks, pre(construct_c_Environment), id_25, i(param(m_Two_tanks,
    construct_c_Environment, arg_instance)) : c_Environment dot s(all_instances) minus
    c_Environment dot v(instances)).
predicat(m_Two_tanks, post(construct_c_Environment), id_26, c_Environment dot p(instances
    ) = (c_Environment dot v(instances)) \/ {i(param(m_Two_tanks,
    construct_c_Environment, arg_instance))}).
predicat(m_Two_tanks, post(construct_c_Environment), id_27, c_Environment dot p(
    currentInstance) = i(param(m_Two_tanks, construct_c_Environment, arg_instance))).

%Declaration of the attribute highFlowOnce
declaration(id_28, c_Environment, variable, v(att_highFlowOnce), set(pair(atom, atom)), {
    none, ins_environment} --> {lit_true, lit_false}).
predicat(c_Environment, invariant, id_44, c_Environment dot v(att_highFlowOnce) :
    c_Environment dot v(instances) --> m_Two_tanks dot s(e_boolean)).

%Initialisation of the attribute highFlowOnce of the class Environment
predicat(c_Environment, initialisation, id_45, c_Environment dot v(att_highFlowOnce) = {(
    ins_environment /-> lit_false)}).
predicat(m_Two_tanks, post(construct_c_Environment), id_46, c_Environment dot p(
    att_highFlowOnce) = c_Environment dot v(att_highFlowOnce) \/ {i(param(m_Two_tanks,
    construct_c_Environment, arg_instance)) /-> lit_false}).
```

Figure B.13: Generated BZP Code of the Environment Block

```
%%TRANSLATION OF THE STATEMACHINE EnvironmentStateMachine_Region1

%%Declaration of the set of states
 declaration(id_73, c_Environment, static,
  s(set_of_states_EnvironmentStateMachine_Region1), set(atom),
  {st_NoFlowState, st_HighFlowState, st_LowFlowState, st_Initial1}).

 predicat(c_Environment, static, id_107,
  c_Environment dot s(set_of_states_EnvironmentStateMachine_Region1) =
  {st_NoFlowState, st_HighFlowState, st_LowFlowState, st_Initial1}).

%%Declaration of the variable status
 declaration(id_74, c_Environment, variable, v(status), set(pair(atom, atom)),
  {none, ins_environment} <-> {st_NoFlowState, st_HighFlowState,
  st_LowFlowState, st_Initial1}).

 predicat(c_Environment, invariant, id_108,
  c_Environment dot v(status) : c_Environment dot v(instances) <->
  c_Environment dot s(set_of_states_EnvironmentStateMachine_Region1)).

 predicat(c_Environment, initialisation, id_109,
  c_Environment dot v(status) = {(ins_environment |-> st_Initial1)}).

 predicat(m_Two_tanks, post(construct_c_Environment), id_110,
  st_Initial1 dot p(status) @ i(param(m_Two_tanks, construct_c_Environment, arg_instance))
  = lit_true).
```

Figure B.14: Generated BZP Code of the Environment State machine

```
%%% Declaration of the state LowFlowState
context(st_LowFlowState).

%%% Declaration of the variable status
declaration(id_77, st_LowFlowState, variable, v(status),
 set(pair(atom, atom)), {none, ins_environment} +-> {lit_true, lit_false}).

predicat(c_Environment, invariant, id_78,
 st_LowFlowState dot v(status) : c_Environment dot v(instances) -->
 m_Two_tanks dot s(e_boolean)).

predicat(c_Environment, initialisation, id_79,
 st_LowFlowState dot v(status) = {(ins_environment /-> lit_false)}).

% Declaration of the action onEntry of the state LowFlowState
operation(id_80, st_LowFlowState, entry).

% Declaration of the postCondition (on status) of the action onEntry of the state
    LowFlowState
predicat(st_LowFlowState, post(entry), id_114,
 st_LowFlowState dot p(status) @ c_Environment dot v(currentInstance) = lit_true).

% Declaration of the action onExit of the state LowFlowState
operation(id_81, st_LowFlowState, exit).

% Declaration of the preCondition (on status) of the action onExit of the state
    LowFlowState
predicat(st_LowFlowState, pre(exit), id_115,
 st_LowFlowState dot v(status) @ c_Environment dot v(currentInstance) = lit_true).

% Declaration of the postCondition (on status) of the action onExit of the state
    LowFlowState
predicat(st_LowFlowState, post(exit), id_116,
 st_LowFlowState dot p(status) @ c_Environment dot v(currentInstance) = lit_false).

%Ficticious transition for state LowFlowState
event(id_117, st_LowFlowState, tr_stopRunToCompletion_st_LowFlowState).

predicat(st_LowFlowState, pre(tr_stopRunToCompletion_st_LowFlowState),
 id_118, st_LowFlowState dot v(status) @ c_Environment dot v(currentInstance) =
 lit_true & c_Environment dot v(opCalled) = none &
 m_Two_tanks dot v(runToCompletion) = lit_true).

predicat(st_LowFlowState, post(tr_stopRunToCompletion_st_LowFlowState),
 id_119, m_Two_tanks dot p(runToCompletion) = lit_false).
```

Figure B.15: Generated BZP Code of the LowFlowState State

```
% External transition LowFlowState to HighFlowState
event(id_128, st_LowFlowState, tr_tr_low_high).

predicat(st_LowFlowState, pre(tr_tr_low_high), id_129, st_LowFlowState dot v(status)
 @ c_Environment dot v(currentInstance) = lit_true & 1=1
 & c_Environment dot v(opCalled) = b_op_activateHighFlowLevel).

predicat(st_LowFlowState, post(tr_tr_low_high), id_130, exe(st_LowFlowState dot exit) &
 c_Environment dot p(att_highFlowOnce) @ c_Environment dot v(currentInstance) = lit_true
     &
 exe(st_HighFlowState dot entry) & c_Environment dot p(opCalled) = none &
 m_Two_tanks dot p(runToCompletion) = lit_true).
```

Figure B.16: Generated BZP Code of the External Transition LowFlowState to HighFlowState

Document generated with LaTeX and:
the LaTeX style for PhD Thesis created by S. Galland — http://www.multiagent.fr/ThesisStyle
the tex-upmethodology package suite — http://www.arakhne.org/tex-upmethodology/

Abstract:

The research conducted during this thesis fall within the scope of modeling, verification and validation of critical and complex systems. This work aims to bridge the gap between the abstract high-level models, starting point of the MBSE process (Model-Based Systems Engineering), and real-time simulation keystone of In-the-Loop processes. In this context, we propose to unify, within a SysML model, continuous aspects of a system, to automatically generate an executable Modelica model (simulation), and discrete aspects allowing animation and test generation by constraint solvers. The work done during this thesis allowed the study and the realization of an original tooled approach to simulate and test such systems from SysML models within a In-the-Loop context. This approach has been validated by two concrete case studies from research partners. The first, from the ANR Smart Blocks project, allowed us to assess the relevance of the proposed SysML modeling methodology in order to perform contact less conveyor simulations. The second case study, from the GEOSEFA Regional project has allowed us to validate the overall approach (simulation and testing) in a In-the-Loop context. It covers the design and the validation of a new energy hybrid system embedded in a helicopter.

Keywords: SysML, Validation, Simulation, Animation, Test generation, MBSE, MDA

Résumé :

Les travaux de recherche menés au cours de cette thèse s'inscrivent dans le cadre de la modélisation, de la vérification et de la validation de systèmes complexes, critiques et multi-physiques. Ces travaux visent à combler l'écart d'abstraction entre les modèles haut-niveau, point de départ des processus MBSE (Model-Based Systems Engineering), et la simulation temps réel, clef de voûte des approches In-the-Loop. Dans ce contexte, nous proposons d'unifier, au sein d'un même modèle SysML, les aspects continus d'un système, permettant de générer de manière automatique un modèle Modelica de plus bas niveau directement exécutable (simulation), et les aspects discrets, permettant l'animation et la génération de tests par des solveurs de contraintes. Les travaux réalisés au cours de cette thèse ont permis l'étude et la réalisation d'une chaîne outillée originale permettant de simuler et de tester ce type de systèmes à partir de modèles SysML en contexte In-the-Loop. Cette démarche a été validée par deux cas d'étude concrets issus de la recherche. Le premier, issu du projet ANR Smart Blocks, nous a permis de mettre à l'épreuve la méthodologie de modélisation SysML dans le but d'effectuer des simulations de convoyeur sans contact (jets d'air). Le second cas d'étude, issu du projet Région GEOSEFA, nous a permis de valider l'approche complète (simulation et test) en contexte In-the-Loop. Celui-ci porte sur la conception et la validation d'un nouveau système énergétique hybride embarqué dans un hélicoptère.

Mots-clés : SysML, Validation, Simulation, Animation, Génération de tests, MBSE, MDA

SPIM

■ École doctorale SPIM 16 route de Gray F - 25030 Besançon cedex
■ tél. +33 (0)3 81 66 66 02 ■ ed-spim@univ-fcomte.fr ■ www.ed-spim.univ-fcomte.fr

UFC
UNIVERSITÉ DE FRANCHE-COMTÉ